

版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



云计算与虚拟化技术丛书

PACKT
UBLISHING

Mastering Elasticsearch
Second Edition

深入理解 Elasticsearch

(原书第2版)

[美] 拉斐尔·酷奇 (Rafał Kuć) 著
马雷克·罗戈任斯基 (Marek Rogoziński)
张世武 余洪淼 商旦 译

资深软件开发专家、架构师撰写，系统且深入阐释Elasticsearch涉及的工具、方法、原则和最佳实践
第2版全面更新，突出Elasticsearch使用过程中的进阶技能，如查询优化、用户体验改善和集群管理等



机械工业出版社
China Machine Press



内 容 简 介

Elasticsearch是一个当前比较流行的分布式开源搜索和数据分析引擎，具备高性能、易扩展、容错性强等特点。它强化了Apache Lucene的搜索能力，把掌控海量数据索引和查询的方式提升到一个新的层次。

本书涵盖Elasticsearch的许多中高级功能，并介绍了缓存、Apache Lucene库以及监控等模块的内部运作机制，提供大量实用案例，如配置Elasticsearch参数、使用监控API等。通过阅读本书，你将学到：

- 了解Apache Lucene和Elasticsearch的设计和架构。
- 配置使用不同的打分模型来改进默认的打分机制。
- 为集群部署选择合适的分片和副本数。
- 使用Elasticsearch提供的功能来改进用户搜索体验。
- 掌握索引段合并，了解Elasticsearch选择段合并的背后机理。
- 开发自定义Elasticsearch插件，并通过多个案例掌握如何编写自己需要的插件。
- 构建高效、可扩展且容错性强的集群，并使用Elasticsearch API来监控集群状态。



华章IT
HZBOOKS | Information Technology





云计算与虚拟化技术丛书

Mastering Elasticsearch
Second Edition

深入理解 Elasticsearch

(原书第2版)

[美]

拉斐尔·酷奇 (Rafał Kuć)

马雷克·罗戈任斯基 (Marek Rogoziński)

著

张世武 余洪淼 商旦 译



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

深入理解 Elasticsearch (原书第 2 版) / (美) 拉斐尔·酷奇 (Rafał Kuć) 等著; 张世武等译. —北京: 机械工业出版社, 2017.5 (2018.1 重印)

(云计算与虚拟化技术丛书)

书名原文: Mastering Elasticsearch, Second Edition

ISBN 978-7-111-56825-4

I. 深… II. ①拉… ②张… III. 互联网络—情报检索 IV. G254.928

中国版本图书馆 CIP 数据核字 (2017) 第 087852 号

本书版权登记号: 图字: 01-2016-6257

Rafał Kuć, Marek Rogoziński: Mastering Elasticsearch, Second Edition (ISBN: 978-1-78355-379-2).

Copyright © 2015 Packt Publishing. First published in the English language under the title “Mastering Elasticsearch, Second Edition”.

All rights reserved.

Chinese simplified language edition published by China Machine Press.

Copyright © 2017 by China Machine Press.

本书中文简体字版由 Packt Publishing 授权机械工业出版社独家出版。未经出版者书面许可, 不得以任何方式复制或抄袭本书内容。

深入理解 Elasticsearch (原书第 2 版)

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 张梦玲

责任校对: 殷虹

印 刷: 三河市宏图印务有限公司

版 次: 2018 年 1 月第 1 版第 2 次印刷

开 本: 186mm × 240mm 1/16

印 张: 20

书 号: ISBN 978-7-111-56825-4

定 价: 79.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

The Translator's Words 译者序

随着互联网时代的来临，人类面临着前所未有的信息过载问题。为了方便人们从海量信息中快速精准地检索感兴趣的内容，Web 搜索引擎应运而生。在互联网发展早期，数据量比较小，单机索引就能支撑起一个完整应用。在这个时期，Apache Lucene 凭借其精巧的代码设计、优异的性能、丰富的查询接口，以及众多衍生搜索产品（如 Apache Solr、Nutch 等），在开源搜索领域大放异彩。随着互联网的发展，数据量快速膨胀，此时对搜索引擎提出了分布式、准实时、高容错、可扩展、易于交互等诸多进阶要求。基于 Lucene 的简单二次开发已经不能满足日常搜索需求，Elasticsearch 的诞生很好地满足了上述大数据时代的搜索产品需求。

Elasticsearch 是一款基于 Apache Lucene 的开源搜索引擎产品，最早发布于 2010 年。之后 Elasticsearch 的开发团队成立了专门的商业公司，继续对其进行开发并提供服务和技术支持。Elasticsearch 具有开源、分布式、准实时、RESTful、便于二次开发等特点，代码实现精巧，系统稳定可靠，已经被国内外众多知名组织和公司广泛采用。

本书内容丰富，不仅深入介绍了 Apache Lucene 的评分机制、查询 DSL、底层索引控制，还详细介绍了 Elasticsearch 的分布式索引机制、用户体验的改善、系统管理及性能优化，以及自定义插件的开发。本书文笔优雅，辅以大量翔实的实例，能帮助读者快速提高 Elasticsearch 应用水平。需要提醒读者的是，本书的目标读者是 Elasticsearch 的中高级用户，如果读者对 Elasticsearch 的基础概念（如 Mapping、Type）缺乏了解的话，建议先阅读作者另一本针对初学者的书《Elasticsearch Server》（目前已经出了第 2 版）。

相对于第 1 版，本书重新组织了多个章节的内容，删减了部分对读者来说意义不大的内容，如 Java API 的使用，突出了 Elasticsearch 使用过程中的进阶技能，比如查询优化、用户体验改善、Elasticsearch 节点集群管理等。

本书译文经过精心组织，结合了译者对 Elasticsearch 的使用经验，并参考了 IBM、微软、百度、腾讯等众多知名企业的业界专业人士的意见。本书翻译团队由拥有丰富 Elasticsearch 实践经验的搜索算法专家、架构师和开发者组成。其中，张世武负责第 3、5、6 章的翻译和校对工作，余洪森负责第 7、8、9 章的翻译和校对，商旦负责第 1、2、4 章的翻译及全书校对。在

IV

本书翻译过程中，翻译团队经过多次讨论、审校，力求翻译准确、优雅。由于本书涉及很多新概念，业界尚无统一术语，另外译者水平有限，难免会出现一些问题，欢迎广大读者及业内同行批评指正。

翻译团队的所有成员在精力和时间都很有限的情况下，能够四个月如一日，顶着日常工作和生活的压力，不计回报地通力合作，为一本书的翻译付出极大的热忱，实在难得。这里要感谢翻译团队家人们的理解和默默支持，让我们一路走到今天。

在本书的翻译过程中，译者张世武得到所在单位成都数联铭品科技有限公司（BBD）的大力支持。BBD 是一家行业领先的金融大数据公司，Elasticsearch 在 BBD 得到了广泛的应用。感谢曾途、尹康、丁国栋、宋开发、刘荀、田志伟、何宏靖等领导对本书出版给予的关怀与支持。技术部及兄弟部门同事提出了很多宝贵建议，感谢以下同事的支持：李想、王尧、丁明会、范东来、刘世林、赵龙、王振宇、何耀、刘兆强、金涛、黄光虎、范从俊、许尧、张伟、邢杰、丁永强、程柳航、何俊、唐国海、刘兴洋、苏印、刘俊杰、闫俊杰、张学锋、徐胜、吕司君、吴德文、邹俊、何正开、刘龙均、郭羽凌、吴勇、杨熹岑、曾昌强、秦德强、谭亚军、魏犇、席爱龄、沈思丞、田曲、陈珊、胡馨月、申秋艳、马小思、高翔、葛相宇、唐斌、苏柏铨、魏林玮、沈思成、韩远、夏阳、杨建蓉、孙健、蒲雪芹、何晓宁、薛双凯、张冬冬。

同样需要感谢机械工业出版社的编辑团队，感谢他们提供的这次翻译机会。尤其需要感谢张梦玲女士，没有她的督导和支持，本书很难如期翻译校对完毕。还要感谢出版社相关岗位的工作人员，离开他们的支持，本书也无法很快与读者见面。

再次感谢！

译者

2017 年 3 月

About the Author 作者简介

Rafał Kuć 是一个很有天赋的团队领袖以及软件开发人员。他现在担任 Sematext 集团公司的咨询专家及软件工程师，专注于开源技术，如 Apache Lucene、Solr、Elasticsearch、Hadoop stack 等。他拥有超过 13 年的软件研发经验，涉及领域广阔，从银行软件到电子商务产品，主要侧重于 Java 平台。除此之外，他对能提高研发效率的任意工具或编程语言都抱有极高的热情。同时他也是 solr.pl 网站的创始人之一，该网站致力于帮助人们解决 Solr、Lucene 相关问题。他还是世界范围内各种会议热邀的演讲嘉宾，之前受邀出席过 Lucene Eurocon、Berlin Buzzwords、ApacheCon、Lucene Revolution、DevOps Days 等会议。

早在 2002 年 Rafał 就接触到 Lucene，一开始他并不喜欢这个开源产品，然而在 2003 年，当他再次使用 Lucene 时，他改变了自己的看法，并看到了搜索技术的巨大潜力。随后 Solr 诞生了。Rafał 于 2010 年开始使用 Elasticsearch。Rafał 目前主要着眼于 Lucene、Solr、Elasticsearch 和信息检索等方面的研究。

Rafał 是《Solr 3.1 Cookbook》一书及其后续版本《Solr 4.0 Cookbook》以及再版作品《Solr Cookbook, Third Edition》的作者。他也是《Elasticsearch Server》第 1 版和第 2 版、《Mastering Elasticsearch》第 1 版的作者，这些书均由 Packt Publishing 出版。

随着 Elasticsearch 的快速发展，我和 Marek 考虑针对本书第 1 版做一次更新。这本书并不适合所有读者，不过在第 1 版中我们没有做出足够的强调。我们把本书第 1 版作为《Elasticsearch Server》的升级版。本书第 1 版的定位也是如此：此刻你手捧的这本书是对《Elasticsearch Server, Second Edition》的扩展和续作。正因为如此，本书可以聚焦在一些高级主题上，比如如何选择恰当的查询方式、如何扩展 Elasticsearch、大量的评分细节和示例、过滤器内部机制、新增的聚集器、不同文档关系处理方式的优劣对比，等等。期望在读完本书后，读者能够更好地理解 Elasticsearch 和 Apache Lucene 底层架构的各种细节，这将有助于他们更方便快捷地获取所需知识。

在此，我想感谢我的家庭，当我在电脑屏幕前全身心投入本书写作的那些日日夜夜里，他们表现出极大的耐心，是我最坚强的后盾。

VI

同样也要感谢 Sematext 所有的同事们，尤其是 Otis，感谢他为我付出的时间，并让我深刻认识到 Sematext 是一个非常适合我的公司。

最后，非常诚挚地感谢 Elasticsearch、Lucenec 项目的所有创建者和开发者，感谢他们杰出的工作和对开源项目的热情。没有他们，就没有本书的诞生；没有他们，开源搜索引擎就不会有现在的这种活力。

再次感谢你们！

——Rafał Kuć

Marek Rogoziński 是一个有着超过 10 年经验的软件架构师和咨询师。其技术专长涉及基于开源搜索引擎（如 Solr、Elasticsearch 等）的解决方案及大数据分析技术（Hadoop、HBase、Twitter Storm 等）。

他也是 solr.pl 网站的联合创始人之一，该网站致力于提供 Solr 和 Lucene 相关资讯和教程。他同时也是 Packt Publishing 出版的《Elasticsearch Server》第 1 版和第 2 版、《Mastering Elasticsearch》第 1 版^①的作者之一。

目前，他是 ZenCard 公司的首席技术官和主架构师。ZenCard 公司提供实时处理和分析大规模交易事务的相关工具，可自动化、匿名识别所有零售渠道（移动电子商务、电子商务、实体店）的零售客户，给零售商提供客户留存和忠诚度运营的工具。

这已经是我们完成的第 4 本关于 Elasticsearch 的书了。在编写本书时，我一再沉醉于 Elasticsearch 的快速进化。我们一直在被标记为“试验中”和“正在开发中”的各种特性之间求取平衡，也不得不冒着最终代码变更，甚至某些有趣的功能被废弃的风险写作相关内容。本书第 2 版重写了很多内容，并引入不少 Elasticsearch 的新特性。不过，这些更新也是有代价的：我们剔除了一些对读者来说不是那么重要的信息。在本书中，我们还试图为读者介绍一些 Elasticsearch 相关的扩展主题。不过请注意，Elasticsearch 的完整生态系统和 ELK 开发栈（Elasticsearch、Logstash 和 Kibana）或者 Hadoop 集成，都需要精心撰写单独的书籍加以介绍。

现在，致谢时间到了。

感谢 Elasticsearch、Lucene 及所有相关产品的创造者们。

同时也要感谢本书的出版团队。尤其是感谢帮助检查错误、校稿、消除表达歧义的伙伴们。非常感谢给我们留言评论和提出建设性意见的读者们。发现有人认可我们的工作令我感到非常惊喜，并深受鼓舞。谢谢你们！

最后，感谢所有支持我和理解我写作的朋友们。

——Marek Rogoziński

① 《Mastering Elasticsearch》的中文版《深入理解 Elasticsearch》已由机械工业出版社出版，书号为 978-7-111-52416-8，本书为《Mastering Elasticsearch, Second Edition》的中文版。——编辑注

About the Reviewer 评审者简介

Hüseyin Akdoğan 的软件职业生涯起源于 GwBasic 编程语言。他先后掌握了 QuickBasic 和 Visual Basic 语言，并用 VB 开发了许多应用程序。2000 年，他迈入了 Web 和 PHP 的世界，后来又穿插使用 Java 语言进行开发。从 2005 年起，他一边进行咨询和培训活动，一边继续使用 Java EE 技术开发企业应用程序。他擅长 JavaServer Faces、Spring Frameworks，以及诸如 NoSQL 和 Elasticsearch 等大数据技术。除此之外，他还在钻研其他大数据相关技术。

Julien Duponchelle 是一位毕业于欧洲理工学院的法国工程师。他致力于开发提升 IT 团队工作效率的各种工具，并在其职业生涯中贡献了多个开源软件项目。

自从他加入 ETNA（一个法国 IT 职业教育学校）开始涉足教育领域起，Julien 以首席支持工程师的身份协助过好几个创业公司进行工作，并参与了多个重大项目的募资活动，包括 Plizy 和 Youboox。

感谢我的女友 Maëlig，感谢她在我评审本书和开发其他开源项目的无数个夜晚的善意支持和极大的耐心。

——Julien Duponchelle

Marcelo Ochoa 现任教于阿根廷布宜诺斯艾利斯省中部国立大学精确科学与自然科学学院的系统实验室，也是 Scotas.com 公司的 CTO，该公司致力于提供基于 Solr 和 Oracle 的准实时搜索解决方案。在高校任职的同时，他也参与一些与 Oracle、大数据相关的外部项目。他参与过的 Oracle 相关项目有 Oracle 手册文档翻译、多媒体培训等。其技术背景涉及数据库、网络、Web、Java 等。在 XML 领域，他参与过 Apache Cocoon 中的 DB Generator，开源项目 DBPrism、DBPrism CMS，基于 Oracle JVM Directory 的 Lucene-Oracle 集成方案，Restlet.org 项目中的 Oracle XDB Restlet Adapter（一个能在基于数据库驻存的 JVM 内部生成本地 REST Web 服务的解决方案）等项目或模块的开发，并因此为业界所熟知。

从 2006 年开始，他参与了 Oracle ACE 计划，这是 Oracle 公司官方推出的一个计划，旨在认可和奖励 Oracle 技术社区中技术娴熟并愿意分享他们的知识、经验的成员为该社区所做



的贡献。

Marcelo Ochoa 还是《Oracle Database Programming Using Java and Web Services》(Kuassi Mensah, 由 Digital Press 出版)和《Professional XML Databases》(Kevin Williams, 由 Wrox Press 出版)两书的合著者,同时也是 Packt 出版社的多部书籍,如《Apache Solr 4 Cookbook》《Elasticsearch Server》等的评审者。



Preface 前言

欢迎来到 Elasticsearch 的世界并阅读本书第 2 版。通过阅读本书，我们将带领你接触与 Elasticsearch 紧密相关的各种话题。请注意，本书不是为初学者写的。笔者将本书作为《Elasticsearch Server, Second Edition》的续作和姊妹篇。相对于《Elasticsearch Server》，本书涵盖了很多新知识，不过你偶尔也可以在本书中发现一些引自《Elasticsearch Server》的内容。

本书将探讨与 Elasticsearch 和 Lucene 相关的多个不同主题。首先，我们以介绍 Lucene 和 Elasticsearch 的基本概念作为开始，带领读者认识 Elasticsearch 提供的众多查询方式。在这里，将涉及和查询相关的不同主题，比如结果过滤以及如何为特定场景选择合适的查询方式。显然，Elasticsearch 不仅仅只有查询功能。因此，本书还将介绍 Elasticsearch 新加入的聚集功能，以及众多能够赋予被索引数据意义的特性，并设法提供更佳的用户查询体验。

对大多数用户来说，查询和数据分析是 Elasticsearch 最吸引人的部分，不过这些还不是我们想要探索的全部内容。因此，本书在涉及索引架构时还会试图跟读者探讨一些额外话题，比如如何选择合适的分片数和副本数，如何调整分片分配行为等。当谈论 Elasticsearch 和 Lucene 之间的关系时，我们还将介绍不同的打分算法、算法之间的差异、如何选择合适的存储机制，以及为什么需要做此选择。

最后，我们还将触及 Elasticsearch 的管理功能，将探讨发现和恢复模块，以及对人类友好的 Cat API。Cat API 可以帮助我们快速获取相关的运维信息，它的返回数据组织成一种大多数人都易于阅读的格式，无需进行 JSON 解析。我们还将认识和使用部落节点，它能够为我们提供在多个节点间联合查询的能力。

因为本书的书名，我们无法忽略与性能相关的话题，所以我们决定用整整一章来探讨性能。我们谈论了文档取值及其相关改进，还介绍了垃圾回收器的工作方式，以及在垃圾回收器未能如我们期望般工作时可以做什么。最后，探讨了如何扩展 Elasticsearch 以应对高索引量和查询量的场景。

和本书第 1 版一样，我们决定以开发 Elasticsearch 插件的话题作为本书结尾。我们将展示如何构建 Apache Maven 项目，并开发两个不同类型的插件——自定义 REST 操作插件和自定义



分析插件。

假如你在读完某些主题后对其产生浓厚的兴趣，那么这本书就是适合你的。希望你在读完后能够喜欢这本书。

本书主要内容

第1章先介绍 Apache Lucene 的工作方式，再介绍 Elasticsearch 的基本概念，并演示 Elasticsearch 内部是如何工作的。

第2章描述 Lucene 评分过程，为什么要进行查询改写，什么是查询模板以及如何使用查询模板。除此之外，还介绍了过滤器的使用，以及如何为特定场景选择合适的查询方式。

第3章描述了查询二次评分、多匹配控制，并介绍了用于做查询分析的各种聚合类型。关键词项聚合和最优词项聚合可以根据所含内容片段对文档进行归类。除此之外，还介绍了 Elasticsearch 的 parent-child 文档关系处理，并提供了在 Elasticsearch 中使用脚本的相关知识。

第4章覆盖了有关用户体验提升的相关话题。本章介绍了查询建议 (suggester)，它能帮助修正查询中的拼写错误并构建高效的自动完成 (autocomplete) 机制。除此之外，通过实际的案例展示如何通过使用不同查询类型和 Elasticsearch 的其他功能来提高查询相关性。

第5章介绍了以下技术：如何选择合适的分片及副本数，路由是如何工作的，索引分片机制是如何工作的以及如何影响分片行为。同时介绍了什么是查询执行偏好，以及它是如何影响查询执行的。

第6章描述如何修改 Lucene 评分以及如何选择备用的评分算法。本章也介绍了 Elasticsearch 的准实时搜索和索引，事务日志的使用，理解索引的段合并，以及如何调整段合并来适应应用场景。在本章最后，还将介绍 Elasticsearch 的缓存机制和请求打断器，以避免出现内存用尽的故障。

第7章介绍了什么是发现、网关、恢复模块，如何配置这些模块，以及有哪些令人心烦的疑难点。还介绍了什么是 Cat API，如何把数据备份到各种云服务上（比如亚马逊的 AWS 和微软的 Azure），以及如何从云服务上恢复数据。最后还介绍了如何使用部落节点进行联盟搜索。

第8章覆盖了与 Elasticsearch 性能相关的各种主题，从使用文档取值来优化字段数据缓存的内存使用，到 JVM 垃圾回收器的工作原理，再到查询基准测试，最后到如何扩展 Elasticsearch 以适应更高的索引量和查询量场景。

第9章通过演示如何开发你自己的 REST 操作插件和查询语言分析插件来介绍 Elasticsearch 的插件开发。



阅读本书的必备资源

本书写作时采用了 Elasticsearch 的 1.4.x 版本，所有的范例代码应该能在该版本下正常运行。除此之外，读者需要一个能发送 HTTP 请求的命令行工具，例如 curl，该工具在绝大多数操作系统上是可用的。请记住，本书的所有范例都使用了 curl。如果读者想使用其他工具，请注意检查请求的格式，以保证你所选择的工具能正确解析它。

除此之外，为了运行第 9 章的范例，需要读者的机器上已安装了 JDK，并且需要一个编辑器来开发相关代码（或者类似 Eclipse 的 Java IDE）。另外，还要求使用 Apache Maven 进行代码的管理与构建。

本书的目标读者

本书的目标读者是那些对 Elasticsearch 基本概念已经很熟悉但是又想深入了解其本身，同时也对 Apache Lucene、JVM 垃圾收集感兴趣的 Elasticsearch 用户和发烧友。除此之外，想了解如何改进查询相关性、如何使用 Elasticsearch Java API、如何编写自定义插件的读者，也会发现本书的趣味性和实用性。

如果你是 Elasticsearch 的初学者，连查询和索引这些基本概念都不熟悉，那么你会发现本书的绝大多数章节难以理解，因为这些内容假定读者已经有相关背景知识。如果是这种情况，建议参考 Packt 出版社出版的另一本关于 Elasticsearch 的图书——《Elasticsearch Server, Second Edition》。

读者反馈

欢迎读者的任何反馈。请让我们知道你对本书的看法——喜欢或者不喜欢哪些内容。读者的反馈对我们开发一些大众能够受益的课题是非常重要的。

一般性的反馈可直接发送 e-mail 至 feedback@packtpub.com，请在邮件标题中提及相关书名。

如果你觉得自己在某个话题上有专长，并有兴趣撰写一本书或做一些贡献，请浏览 www.packtpub.com/authors 上的作者指南。

客户支持

现在，你已经成为 Packt 图书尊贵的读者了。我们提供了各种服务来帮助你从本次购书行为中获得最大价值。



范例代码下载

如果读者通过 <http://www.packtpub.com> 账号购买了 Packt 图书，可直接在本网站下载范例代码。如果读者在其他地方购买了 Packt 图书，可登录 <http://www.packtpub.com/support>，并注册账号，我们将通过 e-mail 发送代码给你。

勘误

尽管我们已经尽了最大努力来保障图书内容的准确性，错误还是难以避免的。如果读者发现了任何错误，不论是文字部分还是代码部分，请报告我们，我们将不胜感激。你的这种行为能帮助其他读者免受误导，并能帮助我们提高图书后续版本的质量。发现任何错误，请登录 <http://www.packtpub.com/submit-errata>，选择相应的书名，单击 errata submissionform 并在表单中输入错误的详细信息。一旦所提交的错误被确认，它会被上传至我们的网站，或者被添加到现有的勘误表里。可登录 <http://www.packtpub.com/support> 选择具体的书名，查看现有的勘误表。



译者序

作者简介

评审者简介

前言

第 1 章 Elasticsearch 简介.....1

1.1 Apache Lucene 简介.....1

1.1.1 熟悉 Lucene2

1.1.2 Lucene 的总体架构2

1.1.3 分析数据4

1.1.4 Lucene 查询语言5

1.2 何为 Elasticsearch8

1.2.1 Elasticsearch 的基本概念8

1.2.2 Elasticsearch 架构背后的关键

概念10

1.2.3 Elasticsearch 的工作流程10

1.3 在线书店示例14

1.4 小结17

第 2 章 查询 DSL 进阶.....18

2.1 Apache Lucene 默认评分公式解释18

2.1.1 何时文档被匹配上19

2.1.2 TF/IDF 评分公式19

2.1.3 Elasticsearch 如何看评分21

2.1.4 一个例子21

2.2 查询改写24

2.2.1 前缀查询示例24

2.2.2 回到 Apache Lucene26

2.2.3 查询改写的属性28

2.3 查询模板30

2.3.1 引入查询模板31

2.3.2 Mustache 模板引擎33

2.3.3 把查询模板保存到文件35

2.4 过滤器的使用及作用原理36

2.4.1 过滤及查询相关性36

2.4.2 过滤器的工作原理40

2.4.3 性能考量41

2.4.4 后置过滤和过滤查询42

2.4.5 选择正确的过滤方式44

2.5 选择正确的查询方式45

2.5.1 查询方式分类45

2.5.2 使用示例50

2.6 小结65

第 3 章 不只是文本搜索66

3.1 查询二次评分66



3.1.1 什么是查询二次评分.....	67	4.1.3 suggester.....	121
3.1.2 一个查询例子.....	67	4.2 改善查询相关性.....	142
3.1.3 二次评分查询的结构.....	67	4.2.1 数据.....	142
3.1.4 二次评分参数.....	70	4.2.2 改善相关性的探索之旅.....	145
3.1.5 总结.....	70	4.3 小结.....	157
3.2 多匹配控制.....	71		
3.3 重要词项聚合.....	78	第 5 章 分布式索引架构.....	159
3.3.1 一个例子.....	79	5.1 选择合适的分片和副本数.....	159
3.3.2 选择重要词项.....	81	5.1.1 分片和过度分配.....	160
3.3.3 多值分析.....	81	5.1.2 一个过度分配的正面例子.....	161
3.3.4 额外的配置.....	84	5.1.3 多分片与多索引.....	161
3.3.5 使用限制.....	89	5.1.4 副本.....	161
3.4 文档分组.....	89	5.2 路由.....	162
3.4.1 top_hits 聚合.....	90	5.2.1 分片和数据.....	162
3.4.2 一个例子.....	90	5.2.2 测试路由功能.....	162
3.5 文档关系.....	95	5.2.3 索引时使用路由.....	166
3.5.1 对象类型.....	95	5.2.4 别名.....	169
3.5.2 嵌套文档.....	98	5.2.5 多个路由值.....	169
3.5.3 parent-child 关系.....	99	5.3 调整默认分片的分配行为.....	170
3.5.4 其他解决方案.....	102	5.3.1 部署意识.....	171
3.6 Elasticsearch 各版本中脚本的变化.....	102	5.3.2 过滤.....	173
3.6.1 脚本变迁.....	102	5.3.3 运行时更新分配策略.....	174
3.6.2 Groovy 简单介绍.....	103	5.3.4 确定每个节点允许的总分片数.....	175
3.6.3 全文检索中的脚本.....	108	5.3.5 确定每个物理机器允许的 总分片数.....	175
3.6.4 Lucene 表达式.....	115	5.4 查询执行偏好.....	179
3.7 小结.....	118	5.5 小结.....	181
第 4 章 改善用户搜索体验.....	119	第 6 章 底层索引控制.....	182
4.1 改正用户拼写错误.....	119	6.1 改变 Apache Lucene 的评分方式.....	182
4.1.1 测试数据.....	120	6.1.1 可用的相似度模型.....	183
4.1.2 深入技术细节.....	121		



6.1.2 为每字段配置相似度模型	183
6.1.3 相似度模型配置	184
6.1.4 选择默认的相似度模型	185
6.2 选择适当的目录实现——store 模块	188
6.3 准实时、提交、更新及事务日志	191
6.3.1 索引更新及更新提交	192
6.3.2 事务日志	193
6.3.3 准实时读取	194
6.4 控制索引合并	195
6.4.1 选择正确的合并策略	196
6.4.2 合并策略配置	197
6.4.3 调度	199
6.5 关于 I/O 调节	200
6.5.1 控制 I/O 节流	200
6.5.2 配置	200
6.6 理解 Elasticsearch 缓存	202
6.6.1 过滤器缓存	203
6.6.2 字段数据缓存	204
6.6.3 查询分片缓存	212
6.6.4 使用 circuit breaker	213
6.6.5 清除缓存	214
6.7 小结	215

第 7 章 管理 Elasticsearch

7.1 发现和恢复模块	216
7.1.1 发现模块的配置	217
7.1.2 主节点	218
7.1.3 网关和恢复模块的配置	223
7.1.4 索引恢复 API	226
7.2 使用人类友好的 Cat API	229
7.2.1 基础知识	230

7.2.2 使用 Cat API	231
7.2.3 一些例子	232
7.3 备份	232
7.4 联盟搜索	236
7.4.1 测试用的集群	236
7.4.2 建立部落节点	237
7.4.3 通过部落节点读取数据	238
7.4.4 通过部落节点写入数据	239
7.4.5 处理索引冲突	240
7.4.6 屏蔽写操作	241
7.5 小结	242

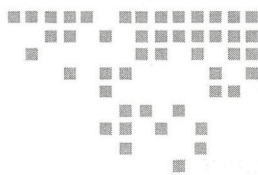
第 8 章 提高性能

8.1 使用 doc values 来优化查询	243
8.1.1 字段缓存存在的问题	244
8.1.2 使用 doc values 的例子	245
8.2 了解垃圾回收器	247
8.2.1 Java 内存	248
8.2.2 解决垃圾回收问题	249
8.2.3 在类 UNIX 系统上避免内存 交换	254
8.3 对查询做基准测试	255
8.3.1 为基准测试配置集群	256
8.3.2 进行基准测试	256
8.3.3 控制运行中的基准测试	259
8.4 热点线程	261
8.4.1 热点线程的使用说明	261
8.4.2 热点线程 API 的响应	262
8.5 扩展 Elasticsearch	263
8.5.1 垂直扩展	263
8.5.2 水平扩展	264



8.5.3 在高负载的场景下使用 Elasticsearch	271
8.6 小结	283
第9章 开发 Elasticsearch 插件	284
9.1 创建 Maven 项目	284
9.2 了解基本知识	285
9.2.1 Maven Java 项目的结构	285
9.2.2 POM 的理念	285
9.2.3 执行构建过程	286
9.2.4 引入 Maven 装配插件	287
9.3 创建自定义 REST 行为	289
9.3.1 设定	289
9.3.2 实现细节	289
9.4 创建自定义分析插件	295
9.4.1 实现细节	295
9.4.2 测试自定义分析插件	302
9.5 小结	304





Elasticsearch 简介

我们希望读者通过阅读本书来获取和拓展关于 Elasticsearch 的基本知识。假设读者已经知道如何使用 Elasticsearch 进行单次或批量索引创建，如何发送请求检索感兴趣的文档，如何使用过滤器缩减检索返回文档的数量，以及使用切面 / 聚合（faceting/aggregation）机制来计算数据的一些统计量。不过，在接触 Elasticsearch 提供的各种令人激动的功能之前，希望读者能对 Apache Lucene 有一个快速的了解，因为 Elasticsearch 使用开源全文检索库 Lucene 进行索引和搜索。此外，我们还希望读者能了解 Elasticsearch 的一些基础概念。为了加速我们的学习，需要牢记这些基础知识，当然，这并不难掌握。同时，我们也需要确保读者能按 Elasticsearch 所需要的那样正确地理解 Lucene。到本章结束为止，将涵盖以下内容：

- Apache Lucene 是什么
- Lucene 的整体架构
- 文本分析过程是如何实现的
- Apache Lucene 的查询语言及其使用
- Elasticsearch 的基本概念
- Elasticsearch 内部是如何通信的

1.1 Apache Lucene 简介

为了全面理解 Elasticsearch 的工作原理，尤其是索引和查询处理环节，对 Apache



Lucene 的理解显得至关重要。揭开 Elasticsearch 神秘的面纱，你会发现它在内部使用 Apache Lucene 创建索引，同时也使用 Apache Lucene 进行搜索。在接下来的几页里，将向读者展示 Apache Lucene 的基本概念，特别是那些从来没有使用过 Lucene 的读者们。

1.1.1 熟悉 Lucene

读者也许会好奇，为什么 Elasticsearch 的创始人决定使用 Apache Lucene 而不是开发一个自己的全文检索库。对于这个问题，笔者并不是很确定，毕竟我们不是这个项目的创始人，我们猜想是因为 Lucene 的以下特点而得到了创始人的青睐：成熟，高性能，可扩展，轻量级以及强大的功能。Lucene 内核可以创建为单个 Java 库文件，并且不依赖第三方代码，用户可以使用它提供的各种所见即所得的全文检索功能进行索引和搜索操作。当然，Lucene 还有很多扩展，它们提供了各种各样的功能，例如多语言处理、拼写检查、高亮显示等。如果不需要这些额外的特性，可以下载单个的 Lucene core 库文件，直接在应用程序中使用它。

1.1.2 Lucene 的总体架构

尽管我们可以直接探讨 Apache Lucene 架构的细节，但是有些概念还是需要提前了解的，以便于更好地理解 Lucene 的架构，它们包括：

- ❑ 文档 (document)：索引与搜索的主要数据载体，它包含一个或多个字段，存放将要写入索引的或将从索引搜索出来的数据。
- ❑ 字段 (field)：文档的一个片段，它包括字段的名称和字段的内容两个部分。
- ❑ 词项 (term)：搜索时的一个单位，代表了文本中的一个词。
- ❑ 词条 (token)：词项在字段文本中的一次出现，包括词项的文本、开始和结束的偏移以及词条类型。

Apache Lucene 将写入索引的所有信息组织为倒排索引 (inverted index) 的结构形式。倒排索引是一种将词项映射到文档的数据结构，它与传统的关系数据库的工作方式不同。你可以认为倒排索引是面向词项的而不是面向文档的。我们来看看简单的倒排索引是什么样的。例如，假设我们有一些只包含 title 字段的文档，如下所示：

- ❑ Elasticsearch Server (文档 1)
- ❑ Mastering Elasticsearch (文档 2)
- ❑ Apache Solr 4 Cookbook (文档 3)

这些文档索引好以后，可简略地显示如下图：



词项	数量	文档
4	1	<3>
Apache	1	<3>
Cooking	1	<3>
Elasticsearch	2	<1><2>
Mastering	1	<1>
Server	1	<1>
Solr	1	<3>

正如你所见，每个词项指向该词项所出现过的文档数。这种索引组织方式允许快速有效的搜索操作，例如基于词项的查询。除了词项本身以外，每个词项有一个与之关联的计数（即文档频率），该计数可以告诉 Lucene 这个词项在多少个文档中出现过。

每个索引由多个段（segment）组成，每个段写入一次但是查询多次。索引期间，一个段创建以后不再修改。例如，文档被删除以后，删除信息单独保存在一个文件中，而段本身并没有被修改。

多个段将会在段合并（segments merge）阶段被合并在一起。或者强制执行段合并，或者由 Lucene 的内在机制决定在某个时刻执行段合并，合并后段的数量更少，但是更大。段合并非常耗费 I/O，合并期间有些不再使用的信息将被清理掉，例如，被删除的文档。对于容纳相同数据的索引，段的数量更少的时候搜索速度更快。尽管如此，还是需要强调一下：因为段合并非常耗费 I/O，请不要强制进行段合并，你只需要仔细配置段合并策略，剩余的事情 Lucene 会自行完成。



注意 如果你想知道段由哪些文件组成以及每个文件都存储了什么信息，请参考 Apache Lucene 的官方文档：http://lucene.apache.org/core/4_10_3/core/org/apache/lucene/codecs/lucene410/package-summary.html。

更深入地了解 Lucene 索引

当然，实际的 Lucene 索引比前面提到的更复杂、更高深，除了词项的文档频率和出现该词项的文档列表外，还包含其他附加信息。在这里我们会介绍一些索引中的附加信息。了解这些信息对我们很有帮助，尽管它们只在 Lucene 内部使用。

（1）norm

norm 是一种与每个被索引文档相关的因子，它存储文档的归一化结果，被用于计算查询的相关得分。norm 基于索引时的文档加权值（boost）计算得出，与文档一起被索引存储。使用 norm 可以让 Lucene 在建立索引时考虑不同文档的权重，不过需要一些额外的磁盘空间和内存来索引和存储 norm 信息。



(2) 词项向量

词项向量 (term vector) 是一种针对每个文档的微型倒排索引。词项向量的每个维由词项和出现频率结对组成, 还可以包括词项的位置信息。Lucene 和 Elasticsearch 默认都禁用词项向量索引, 不过要实现某些功能, 如关键词高亮等需要启用这个选项。

(3) 倒排项格式

随着 Lucene 4.0 的发布, Lucene 引入了解码器架构, 允许开发者控制索引文件写入磁盘的格式, 倒排项就是索引中可定制的部分之一。倒排项中可以存储字段、词项、文档、词项位置和偏移以及载荷 (payload, 一个在 Lucene 索引中随意存放的字节数组, 可以包含任何我们需要的信息)。针对不同的使用目的, Lucene 提供了不同的倒排项格式。比如, 有一种优化后的格式是专门为高散列范围字段如唯一标识提供的。

(4) doc values

我们前面提到过, Lucene 索引是一种倒排索引。不过, 针对某些功能, 如切面 (faceting) 或聚合 (aggregation), 这种倒排索引架构就不是最佳选择。这类功能通常需要操作文档而不是词项, Lucene 需要把索引翻转过来构成正排索引才能完成这些功能所需要的计算。基于这些考虑, Lucene 引入了 doc values 和额外的数据结构来进行分组、排序和聚合。doc values 存储字段的正排索引。Lucene 和 Elasticsearch 都允许我们通过配置来指定 doc values 的存储实现方式。可选的存储实现包括基于内存的、基于硬盘的, 以及二者的混合。

1.1.3 分析数据

读者也许会好奇, 文档中的数据是如何转化为倒排索引的? 查询串又是怎么转换为可以用于搜索的词项的? 这个转换过程被称为分析 (analysis)。

文本分析由分析器来执行, 它建立在分词器 (tokenizer)、过滤器 (filter) 及字符映射器 (character mapper) 之上。

Lucene 的分词器用来将文本切割成词条, 词条是携带各种额外信息的词项, 这些信息包括: 词项在原始文本中的位置, 词项的长度。分词器工作的结果被称为词条流, 因为这些词条被一个接一个地推送给过滤器处理。

除了分词器, 过滤器也是 Lucene 分析器的组成部分。过滤器数额可选, 可以为零个、一个或多个, 用于处理词条流中的词条。例如, 它可以移除、修改词条流中的词条, 甚至可以创造新的词条。Lucene 中有很多现成的过滤器, 你也可以根据需要进行新的过滤器。以下是一些过滤器的例子。

- ❑ 小写过滤器: 将所有词条转化为小写。
- ❑ ASCII 过滤器: 移除词条中所有非 ASCII 字符。
- ❑ 同义词过滤器: 根据同义词规则, 将一个词条转化为另一个词条。



□ 多语言词干还原过滤器：将词条的文本部分归约到它们的词根形式，即词干还原。

当分析器中有多个过滤器时，会逐个处理，理论上可以有无限多个过滤器。

过滤器可以一个接一个地被调用，因此我们可以通过逐个添加多个过滤器的方式来获得近乎无限的分析能力。

最后我们介绍字符映射器，它用于调用分词器之前的文本预处理操作。字符映射器的一个例子就是 HTML 文本的去标签处理。

索引与查询

也许读者会好奇，Lucene 以及所有基于 Lucene 的软件是如何控制索引及查询操作的？在索引期，Lucene 会使用你选择的分析器来处理文档中的内容，可以对不同的字段使用不同的分析器，例如，文档的 title 字段与 description 字段就可以使用不同的分析器。

在检索时，如果你使用了某个查询分析器（query parser），那么你的查询串将会被分析。当然，你也可以选择不分析数据。有一点需要牢记，Elasticsearch 中有些查询会被分析，而有些则不会被分析。例如，前缀查询（prefix query）不会被分析，而匹配查询（match query）会被分析。

你还应该记住，索引期与检索期的文本分析要采用同样的分析器，只有查询（query）分词出来的词项与索引中词项能匹配上，才会返回预期的文档集。例如，如果在索引期使用了词干还原与小写转换，那么在查询期，也应该对查询串做相同的处理，否则，查询可能不会返回任何结果。

1.1.4 Lucene 查询语言

Elasticsearch 提供的一些查询类型（query type）支持 Apache Lucene 的查询解析语法，因此，我们应该深入了解 Lucene 的查询语言。

1. 理解基本概念

在 Lucene 中，一个查询（query）通常被分割为词项与操作符。Lucene 中的词项可以是单个的词，也可以是一个短语（用双引号括起来的一组词）。如果查询被设置为要被分析，那么预先选定的分析器将会对查询中的所有词项进行处理。

一个查询也可以包含布尔操作符。布尔操作符连接多个词项，使之构成从句（clause）。有以下这些布尔操作符。

□ AND：它的含义是，文档匹配当前从句当且仅当 AND 操作符左右两边的词项都在文档中出现。例如，我们想执行“apache AND lucene”这样的查询，只有同时包含“apache”和“lucene”这两个词项的文档才会被返回给用户。

□ OR：它的含义是，包含当前从句中任意词项的文档都被视为与该从句匹配。例如，我们执行“apache OR lucene”这样的查询，任意包含词项“apache”或词项“lucene”的文档都会返回给用户。

□ NOT：它的含义是，与当前从句匹配的文档必须不包含 NOT 操作符后面的词项。例如，我们执行“lucene NOT elasticsearch”这样的查询，只有包含词项“lucene”且不包含词项“elasticsearch”的文档才会被返回给用户。

除了前面介绍的那些操作符以外，我们还可以使用以下这些操作符。

□ +：它的含义是，只有包含了“+”操作符后面词项的文档才会被认为与从句匹配。例如，我们想查找那些必须包含“lucene”但是“apache”可出现可不出现的文档，可执行如下查询：“+lucene apache”。

□ -：它的含义是，与从句匹配的文档，不能出现“-”操作符后的词项。例如，我们想查找那些包含了“lucene”但是不包含“Elasticsearch”的文档，可以执行如下查询：“+lucene -elasticsearch”。

如果查询中没有出现前面提到过的任意操作符，那么默认使用 OR 操作符。

除了前面介绍的内容之外，有一件事情值得一提：可以使用圆括号对从句进行分组，以构造更复杂的从句，例如：

```
Elasticsearch AND (mastering OR book)
```

2. 在字段中查询

就像 Elasticsearch 的处理方式那样，Lucene 中所有数据都存储在字段（field）中，而字段又是文档的组成单位。为了实现针对某个字段的查询，用户需要提供字段名称，再加上冒号以及将要在该字段中执行查询的从句。如果你想查询所有在“title”字段中包含词项“Elasticsearch”的文档，可执行以下查询：

```
title:Elasticsearch
```

也可以在一个字段中同时使用多个从句，例如，如果你想查找所有在“title”字段中同时包含词项“Elasticsearch”和短语“mastering book”的文档，可执行如下查询：

```
title:(+Elasticsearch +"mastering book")
```

当然，上面的查询也可以写成下面这种形式：

```
+title:Elasticsearch +title:"mastering book"
```

3. 词项修饰符

除了使用简单词项和从句的常规字段查询以外，Lucene 允许用户使用修饰符（modifier）修改传入查询对象的词项。毫无疑问，最常见的修饰符就是通配符（wildcard）。Lucene 支持两种通配符：? 和 *。前者匹配任意一个字符，而后者匹配多个字符。



请记住，出于对性能的考虑，通配符不能作为词项的第一个字符出现。

除通配符之外，Lucene 还支持模糊（fuzzy and proximity）查询，办法是使用“~”字符以及一个紧随其后的整数值。当使用该修饰符修饰一个词项时，意味着我们想搜索那些包含该词项近似词项的文档（所以这种查询称为模糊查询）。~字符后的整数值确定了近似词项与原始词项的最大编辑距离。例如，当我们执行查询 `writer ~ 2`，意味着包含词项 `writer` 和 `writers` 的文档都可以被视为与查询匹配。

当修饰符~用于短语时，其后的整数值用于告诉 Lucene 词项之间多大距离是可以接受的。例如，我们执行如下查询：

```
title:"mastering Elasticsearch"
```

在 `title` 字段中包含 `mastering Elasticsearch` 的文档被视为与查询匹配，而包含 `mastering book Elasticsearch` 的文档则被认为不匹配。而如果我们执行下面这个查询：

```
title:"mastering Elasticsearch"~2,
```

则这两个文档都被认为与查询匹配。

此外，还可以使用 ^ 字符并赋以一个浮点数对词项加权（boosting），从而提高该词项的重要程度。如果都被加权，则权重值较大的词项更重要。默认情况下词项权重为 1。可以参考 2.1 节进一步了解什么是权重值（boost value），以及其在文档评分中的作用。

我们也可以使用方括号和花括号来构建范围查询。例如，我们想在一个数值类型的字段上执行一个范围查询，执行如下查询即可：

```
price:[10.00 TO 15.00]
```

上面查询的返回文档的 `price` 字段的值大于等于 10.00 并小于等于 15.00。

当然，我们也可以在字符串类型的字段上执行范围查询（range query），例如：
`name:[Adam TO Adria]`

上面查询的返回文档的 `name` 字段中，包含了按字典顺序介于 Adam 和 Adria 之间（包括 Adam 和 Adria）的词项。

如果想执行范围查询同时又想排除边界值，则可使用花括号作为修饰符。例如，我们想查找 `price` 字段值大于等于 10.00 但小于 15.00 的文档，可使用如下查询：

```
price:[10.00 TO 15.00}
```

如果想执行一边受限而另一边不做限制的范围查询，例如，查找 `price` 字段值大于等于 10.00 的文档，可使用如下查询：

```
price:[10.00 TO *]
```

4. 特殊字符处理

很多应用场景中，也许你想搜索某个特殊字符（这些特殊字符包括 +、-、&&、||、!、(、)、

{ }、[]、^、"、~、*、?、:、\、/), 需要先使用反斜杠对这些特殊字符进行转义。例如, 你可能想搜索 `abc"efg` 这个词项, 需要按如下方式处理: `abc\"efg`

1.2 何为Elasticsearch

当读者手持本书阅读时, 可能已经对 Elasticsearch 有所了解了, 至少已经了解了它的一些核心概念和基本用法。不过, 为了全面理解该搜索引擎是如何工作的, 我们最好简略地讨论一下它。

也许你已经了解到, Elasticsearch 是一个可用于构建搜索应用的成品软件 (译者注: 区别于 Lucene 这种中间件)。它最早由 Shay Banon 创建, 并于 2010 年 2 月发布。之后的几年, Elasticsearch 迅速流行开来, 成为其他开源和商业解决方案之外的一个重要选择。它是下载量最多的开源项目之一。

1.2.1 Elasticsearch 的基本概念

现在, 让我们浏览一下 Elasticsearch 的基本概念以及它们的特征。

1. 索引

Elasticsearch 将它的数据存储在一个或者多个索引 (index) 中。用 SQL 领域的术语来类比, 索引就像数据库, 可以向索引写入文档或者从索引中读取文档。就像之前说过的那样, Elasticsearch 在内部使用 Lucene 将数据写入索引或从索引中检索数据。读者需要注意的是, Elasticsearch 中的索引可能由一个或多个 Lucene 索引构成, 细节由 Elasticsearch 的索引分片 (shard)、复制 (replica) 机制及其配置决定。

2. 文档

文档 (document) 是 Elasticsearch 世界中的主要实体 (对 Lucene 来说也是如此)。对于所有使用 Elasticsearch 的案例来说, 它们最终都会被归结到对文档的搜索之上。文档由字段构成, 每个字段包含字段名以及一个或多个字段值 (在这种情况下, 该字段被称为是多值的, 即文档中有多个同名字段)。文档之间可能有各自不同的字段集合, 文档并没有固定的模式或强制的结构。这种现象看起来很眼熟 (这些规则也适用于 Lucene 文档)。事实上, Elasticsearch 的文档最后都被存储为 Lucene 文档了。从客户端的角度来看, 文档是一个 JSON 对象 (想了解更多关于 JSON 格式的细节, 请参考 <http://en.wikipedia.org/wiki/JSON>)。

3. 类型

Elasticsearch 中每个文档都有与之对应的类型 (type) 定义。这允许用户在一个索

索引中存储多种文档类型，并为不同文档类型提供不同的映射。如果同 SQL 领域类比，Elasticsearch 的类型就像一个数据库表。

4. 映射

正如你在 1.1 节所了解到的那样，所有文档在写入索引前都将被分析。用户可以设置一些参数，决定如何将输入文本分割为词条，哪些词条应该被过滤掉，或哪些附加处理有必要被调用（例如移除 HTML 标签）。这就是映射（mapping）扮演的角色：存储分析链所需的所有信息。虽然 Elasticsearch 能根据字段值自动检测字段的类型，有时候（事实上几乎是所有时候）用户还是想自己来配置映射，以避免出现一些令人不愉快的意外。

5. 节点

单个的 Elasticsearch 的服务实例被称为节点（node）。很多时候部署一个 Elasticsearch 节点就足以应付大多数简单的应用，但是考虑到容错性或者数据膨胀到单机无法应付这些状况，也许你会更倾向于使用多节点的 Elasticsearch 集群。

Elasticsearch 节点可以按用途分为 3 类。众所周知，Elasticsearch 是用来索引和查询数据的，因此第 1 类节点就是数据（data）节点，用来持有数据，提供对这些数据的搜索功能。第 2 类节点指主（master）节点，作为监督者负责控制其他节点的工作。一个集群中只有一个主节点。第 3 类节点是部落（tribe）节点。部落节点是 Elasticsearch 1.0 版新引入的节点类型，它可以像桥梁一样连接起多个集群，并允许我们在多个集群上执行几乎所有可以在单集群 Elasticsearch 上执行的功能。

6. 集群

多个协同工作的 Elasticsearch 节点的集合被称为集群（cluster）。Elasticsearch 的分布式属性使我们可以轻松处理超过单机负载能力的数据量。同时，集群也是无间断提供服务的一种解决方案，即便当某些节点因为宕机或者执行管理任务（例如升级）不可用时，Elasticsearch 几乎是无缝集成了集群功能。在我们看来，这是它胜过竞争对手的最主要优点之一。在 Elasticsearch 中配置一个集群是再容易不过的事了。

7. 分片

正如我们之前提到的那样，集群允许系统存储的数据总量超过单机容量。为了满足这个需求，Elasticsearch 将数据散布到多个物理的 Lucene 索引上去。这些 Lucene 索引被称为分片（shard），而散布这些分片的过程叫做分片处理（sharding）。Elasticsearch 会自动完成分片处理，并且让用户看来这些分片更像是一个大索引。请记住，除了 Elasticsearch 本身自动进行分片处理外，用户为具体的应用进行参数调优也是至关重要的，因为分片的数量在索引创建时就被配置好了，之后无法改变，除非创建一个新索引并重新索引全部数据。

8. 副本

分片处理允许用户推送超过单机容量的数据至 Elasticsearch 集群。副本 (replica) 则解决了访问压力过大时单机无法处理所有请求的问题。思路是很简单的, 为每个分片创建冗余的副本, 处理查询时可以把这些副本当作最初的主分片 (primary shard) 使用。值得注意的是, 副本给 Elasticsearch 带来了更多的安全性。如果主分片所在的节点宕机了, Elasticsearch 会自动从该分片的副本中选出一个作为新的主分片, 因此不会对索引和搜索服务产生干扰。可以在任意时间点添加或移除副本, 所以一旦你有需要, 可随时调整副本的数量。

1.2.2 Elasticsearch 架构背后的关键概念

Elasticsearch 的架构遵循了一些设计理念。开发团队希望这个搜索引擎产品易于使用和扩展, 这些特征在 Elasticsearch 的每个角落里都可以被看到。从架构的视角来看, 有下面这些主要特征:

- ❑ 合理的默认配置, 使得用户在简单安装以后能直接使用 Elasticsearch 而不需要任何额外的调优, 这其中包括内置的发现 (例如, 字段类型检测) 和自动配置功能。
- ❑ 默认的分布式工作模式。每个节点总是假定自己是某个集群的一部分或将是某个集群的一部分。
- ❑ 对等架构 (P2P) 可以避免单点故障。节点会自动连接到集群中的其他节点, 进行相互的数据交换和监控操作。这其中就包括了索引分片的自动复制。
- ❑ 容易扩充新节点至集群, 不论是从数据容量的角度还是数量的角度。
- ❑ Elasticsearch 没有对索引中的数据结构强加任何限制。这允许用户调整现有的数据模型。正如之前我们所描述的那样, Elasticsearch 支持在一个索引中存在多种数据类型, 允许用户调整业务模型, 包括处理文档之间的关联 (尽管这种功能非常有限)。
- ❑ 准实时 (near real time searching) 搜索和版本同步 (versioning)。考虑到 Elasticsearch 的分布式特性, 查询延迟和节点之间临时的数据不同步是难以避免的。Elasticsearch 尝试减少这些问题, 并且提供了额外的机制用于版本同步。

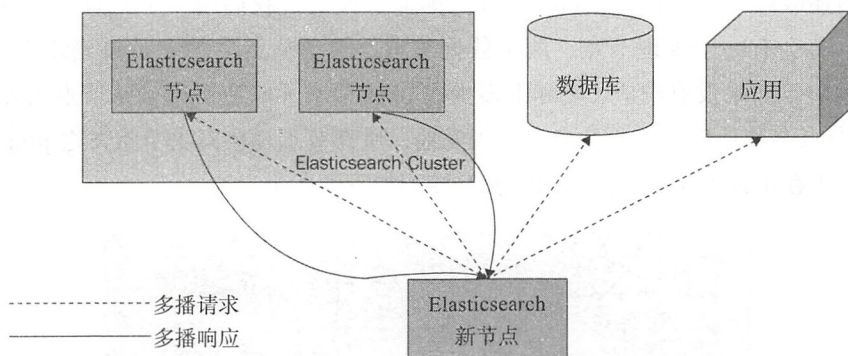
1.2.3 Elasticsearch 的工作流程

本节我们将探索一些关键的 Elasticsearch 特性, 如启动、故障检测、数据索引和查询等。

1. 启动过程

当 Elasticsearch 节点启动时, 它使用发现 (discovery) 模块来发现同一个集群中的其他节点 (这里的关键是配置文件中的集群名称) 并与它们连接。默认情况下, Elasticsearch 节

点会向网络中发送广播请求，以找到拥有相同集群名称的其他节点。读者可以通过下图的描述来了解相关的处理。



集群中有一个节点被选为主 (master) 节点。该节点负责集群的状态管理以及在集群拓扑变化时做出反应，分发索引分片至集群的相应节点上去。



注意

请记住，从用户的角度来看，Elasticsearch 中的管理节点并不比其他节点重要，这与其他某些分布式系统不同（例如数据库）。在实践中，你不需要知道哪个节点是管理节点，所有操作可以发送至任意节点，Elasticsearch 内部会自行处理这些不可思议的事情。如果有需要，任意节点可以并行发送子查询给其他节点，并合并搜索结果，然后返回给用户。所有这些操作并不需要经过管理节点处理（请记住，Elasticsearch 是基于对等架构的）。

管理节点读取集群的状态信息，如果有必要，它会进行恢复 (recovery) 处理。在该阶段，管理节点会检查有哪些索引分片，并决定哪些分片将用作主分片。此后，整个集群进入黄色状态。

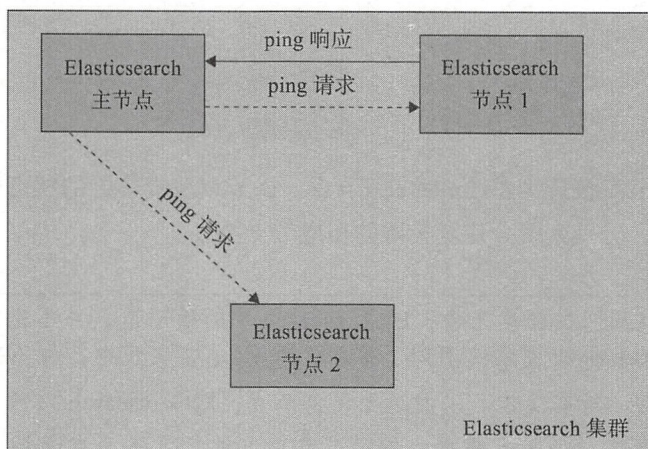
这意味着集群可以执行查询，但是系统的吞吐量以及各种可能的状况是未知的（这种状况可以简单理解为所有的主分片已经被分配了，但是副本没有被分配）。下面的事情就是寻找到冗余的分片用作副本。如果某个主分片的副本数过少，管理节点将决定基于某个主分片创建分片和副本。如果一切顺利，集群将进入绿色状态（这意味着所有主分片以及副本均已分配好）。

2. 故障检测

集群正常工作时，管理节点会监控所有可用节点，检查它们是否正在工作。如果任何节点在预定义的超时时间内不响应，则认为该节点已经断开，然后错误处理过程开始启动。这意味着可能要在集群 - 分片之间重新做平衡，选择新的主节点等。对每个丢失的主分片，

一个新的主分片将会从原来的主分片的副本中选出来。新分片和副本的放置策略是可配置的，用户可以根据具体需求进行配置。更多的信息可以在第 7 章了解到。

为了描述故障检测 (failure detection) 是如何工作的，我们用一个只有 3 个节点的集群作为例子，将会有 1 个管理节点，两个数据节点。管理节点会发送 ping 请求至其他节点，然后等待响应。如果没有响应（实际上多少次 ping 请求无响应可以确认节点失败取决于配置），则该节点会被从集群中移除出去。相反地，所有节点也会向主节点发送 ping 请求来检查主节点是否在正常工作。节点之间的相互探测如下图所示。



3. 与 Elasticsearch 通信

前面已经讨论过 Elasticsearch 是如何构建的了，然而，对普通用户来说，最重要的部分是如何向 Elasticsearch 推送数据以及构建查询。为了提供这些功能，Elasticsearch 对外公开了一个设计精巧的 API。如果我们说，基本上每个 Elasticsearch 功能模块都有一个 API，这将是令人鼓舞的。这个主 API 是基于 REST 的（REST 细节请参考 http://en.wikipedia.org/wiki/Representational_state_transfer），并且在实践中能轻松整合到任意支持 HTTP 协议的系统中去。

Elasticsearch 假设数据由 URL 携带或者以 JSON（JSON 细节请参考 <http://en.wikipedia.org/wiki/JSON>）文档的形式由 HTTP 消息体携带。使用 Java 或者基于 JVM 语言的用户，应该了解一下 Java API，它除了 REST API 提供的所有功能以外还有内置的集群发现功能。

值得一提的是，Elasticsearch 在内部也使用 Java API 进行节点间通信。因此，Java API 提供了所有可被 REST API 调用的功能。

（1）索引数据

Elasticsearch 提供了多种索引数据的方式。最简单的方式是使用索引 API，它允许用户发送一个文档至特定的索引。例如，使用 curl 工具（curl 细节请参考 <http://curl.haxx.se/>），

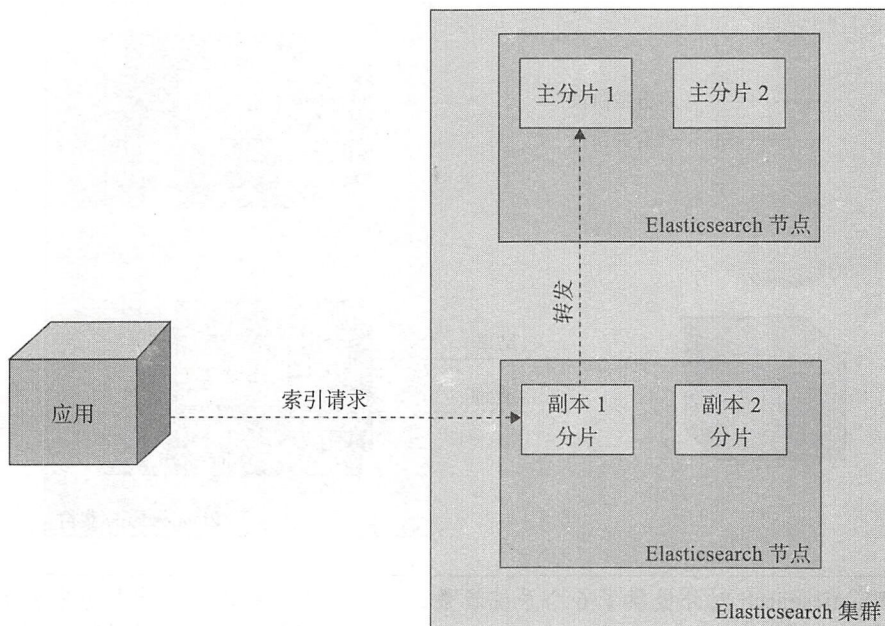
可以使用如下命令创建一个文档：

```
curl -XPUT http://localhost:9200/blog/article/1 -d '{"title": "New
version of Elastic Search released!", "tags": ["announce",
"Elasticsearch", "release"]}'
```

第2种方式允许用户通过 bulk API 或 UDP bulk API 一次发送多个文档至集群。两者的区别在于网络连接方式，前者使用 HTTP 协议，后者使用 UDP 协议。后者速度快，但是不可靠。还有一种方式使用被叫作河流（river）的插件来发送数据。不过在这里我们不需要了解这种河流插件，因为它们将在 Elasticsearch 未来版本中被移除。

有一件事情需要记住，建索引操作只会发生在主分片上，而不是副本上。当一个索引请求被发送至一个节点上时，如果该节点没有对应的主分片或者只有副本，那么这个请求会被转发到拥有正确的主分片的节点。然后，该节点将会把索引请求群发给所有副本，等待它们的响应（这一点可以由用户控制），最后，当特定条件具备时（比如说达到规定数目的副本都完成了更新时）结束索引过程。

下图展示了我们刚刚探讨的索引处理过程。



（2）查询数据

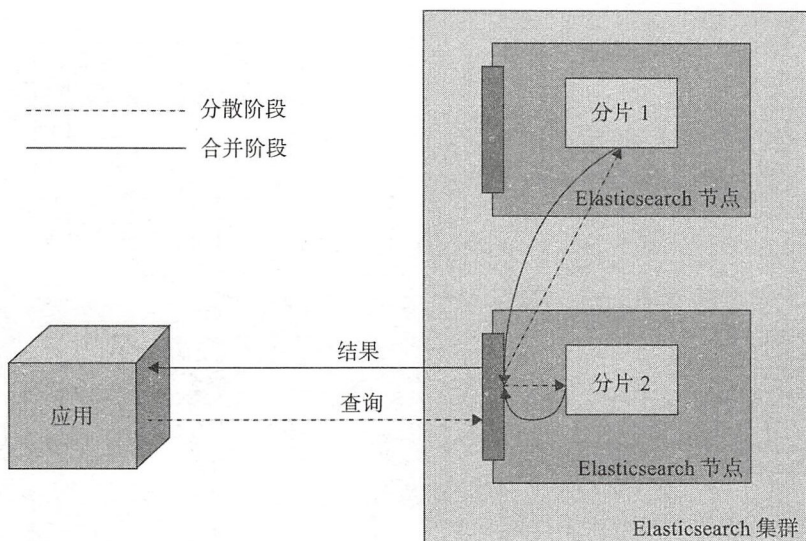
查询 API 占据了 Elasticsearch API 的大部分。使用查询 DSL（基于 JSON 的可用于构建复杂查询的语言），我们可以做下面这些事情：

- ❑ 使用各种查询类型，包括，简单的词项查询，短语查询，范围查询，布尔查询，模糊查询，区间查询，通配符查询，空间查询，以及具备人类可读的打分控制功能的

函数查询，等等。

- ❑ 组合简单查询构建复杂查询。
- ❑ 文档过滤，在不影响评分的前提下抛弃那些不满足特定查询条件的文档。这一点可以提升性能。
- ❑ 查找与特定文档相似的文档。
- ❑ 查找特定短语的查询建议和拼写检查。
- ❑ 使用切面构建动态导航和计算各种统计量。
- ❑ 使用预搜索 (prospective search) 和查找与指定文档匹配的 query 集合。

谈到查询操作，读者应该了解一个很重要的事实：查询并不是一个简单的、单步骤的操作。一般来说，查询分为两个阶段：分散阶段 (scatter phase) 和合并阶段 (gather phase)。在分散阶段将查询分发到包含相关文档的多个分片中去执行查询，而在合并阶段则从众多分片中收集返回结果，然后对它们进行合并、排序，进行后续处理，然后返回给客户端。该机制可以由下图描述。



注意 Elasticsearch 对外提供了 6 个系统参数，通过使用其中之一来定制分散 / 合并机制。在本书的姐妹版《Elasticsearch Server, Second Edition》(Packt 出版社) 中已经讨论过这个问题了。

1.3 在线书店示例

本书可作为《Elasticsearch server, Second Edition》一书的延续。因此，我们在这里也

沿用在那本书中的案例。总的来说，假设自己正在实现和运作一个在线书店。

首先需要有一个 library 索引，它的映射定义如下：

```
{
  "book" : {
    "_index" : {
      "enabled" : true
    },
    "_id" : {
      "index": "not_analyzed",
      "store" : "yes"
    },
    "properties" : {
      "author" : {
        "type" : "string"
      },
      "characters" : {
        "type" : "string"
      },
      "copies" : {
        "type" : "long",
        "ignore_malformed" : false
      },
      "otitle" : {
        "type" : "string"
      },
      "tags" : {
        "type" : "string",
        "index" : "not_analyzed"
      },
      "title" : {
        "type" : "string"
      },
      "year" : {
        "type" : "long",
        "ignore_malformed" : false
      },
      "available" : {
        "type" : "boolean"
      },
      "review" : {
        "type" : "nested",
        "properties" : {
          "nickname" : {
            "type" : "string"
          },
          "text" : {
            "type" : "string"
          },
          "stars" : {
            "type" : "integer"
          }
        }
      }
    }
  }
}
```

```

    }
  }
}
}
}

```

这段映射代码位于随书提供的 `library.json` 文件。

我们将要使用的数据也可以在随书提供的 `books.json` 文件中找到。这个文件中的文档示例如下：

```

{ "index": { "_index": "library", "_type": "book", "_id": "1" } }
{ "title": "All Quiet on the Western Front", "otitle": "Im Westen
nichts
Neues", "author": "Erich Maria Remarque", "year": 1929, "characters":
["Paul
Bäumer", "Albert Kropp", "Haie Westhus", "Fredrich Müller",
"Stanislaus
Katzinsky", "Tjaden"], "tags": ["novel"], "copies": 1, "available":
true,
"section" : 3 }
{ "index": { "_index": "library", "_type": "book", "_id": "2" } }
{ "title": "Catch-22", "author": "Joseph Heller", "year":
1961, "characters":
["John Yossarian", "Captain Aardvark", "Chaplain Tappman", "Colonel
Cathcart", "Doctor Daneeka"], "tags": ["novel"], "copies": 6,
"available" :
false, "section" : 1 }
{ "index": { "_index": "library", "_type": "book", "_id": "3" } }
{ "title": "The Complete Sherlock Holmes", "author": "Arthur Conan
Doyle", "year": 1936, "characters": ["Sherlock Holmes", "Dr. Watson",
"G.
Lestrade"], "tags": [], "copies": 0, "available" : false, "section" :
12 }
{ "index": { "_index": "library", "_type": "book", "_id": "4" } }
{ "title": "Crime and Punishment", "otitle": "Преступление и
наказание", "author": "Fyodor Dostoevsky", "year": 1886, "characters":
["Raskolnikov", "Sofia Semyonovna Marmeladova"], "tags": [], "copies":
0,
"available" : true }

```



读者可以从自己在 <http://www.packtpub.com> 的个人账户中下载所有已购 Packt 书籍的示例代码文件。如果您从其他地方购买本书，可以访问 <http://www.packtpub.com/support> 进行登记，随后 Packt 出版社会把文件通过 e-mail 发送给您。

我们需要执行如下命令来创建带以上映射的索引，并索引数据：

```

curl -XPOST 'localhost:9200/library'
curl -XPUT 'localhost:9200/library/book/_mapping' -d @library.json
curl -s -XPOST 'localhost:9200/_bulk' --data-binary @books.json

```


1.4 小结

在本章中，我们了解了 Apache Lucene 的一般架构，例如它的工作原理，文本分析过程是如何完成的，如何使用 Apache Lucene 查询语言。此外，我们还讨论了 Elasticsearch 的一些基本概念，例如它的基本架构和内部通信机制。

下一章将学习 Apache Lucene 的默认评分公式，什么是查询重写过程（query rewrite process）以及它是如何工作的。除此之外，还将讨论 Elasticsearch 的一些功能，例如查询模板、过滤器，以及它们影响查询性能的机制。我们将学习如何使用过滤器，并选择合适的查询方式来完成查询工作。

查询 DSL 进阶

在上一章，我们了解了什么是 Apache Lucene，它的整体架构，以及文本分析过程是如何完成的。之后，我们还介绍了 Lucene 的查询语言及其用法。除此之外，我们也讨论了 Elasticsearch，讨论了它的架构，以及一些核心概念。在本章，我们将深入研究 Elasticsearch 的查询 DSL (Domain Specific Language)。在了解那些高级查询之前，我们将先了解 Lucene 评分公式的工作原理。到本章结束，将涵盖以下内容：

- Lucene 默认评分公式是如何工作的
- 什么是查询重写
- 什么是查询模板以及如何使用查询模板
- 如何优化复杂的 Boolean 查询
- 复杂 Boolean 查询的性能奥秘
- 如何为特定场景选择合适的查询类型

2.1 Apache Lucene 默认评分公式解释

评分是 Apache Lucene 查询处理过程的一个重要环节。评分是指针对给定查询计算某个文档的 score 属性的过程。什么是文档得分？它是一个刻画文档与查询匹配程度的参数。在本节，我们将了解 Apache Lucene 的默认评分机制：TF/IDF（词频 / 逆文档频率）算法以及它是如何影响文档查询结果的。了解评分公式的工作原理对构造复杂查询以及分析查询中因子的重要性都是很有价值的。同时，掌握 Lucene 评分机制的基础知识有助于我们更好地

优化查询来获取符合我们使用场景的结果。

2.1.1 何时文档被匹配上

一个文档被 Lucene 返回，意味着该文档与用户提交的查询是匹配的。在这种情况下，每个被返回文档会有一个得分。在某些场景下，所有文档的得分都一样（比如使用 `constant_score` 查询），不过一般情况下，各个文档的得分是不一样的。得分越高，文档更相关，至少从 Apache Lucene 及其评分公式的角度来看是这样的。得分还取决于匹配的文档、查询和索引内容，因此，很显然同一个文档对不同查询的得分是不同的。读者需要注意，同一文档在不同查询中的得分不具备可比较性，不同查询返回文档中的最高得分也不具备可比较性。这是因为文档得分依赖多个因子，除了权重和查询本身的结构，还依赖被匹配的词项数目、词项所在字段，以及用于查询规范化的匹配类型，如此等等。在一些比较极端的情况下，同一个文档在相似查询中的得分非常悬殊，仅仅是因为使用了自定义得分查询或者命中词项数的急剧变化。

现在，让我们再回到评分过程。为了计算文档得分，我们需要考虑以下这些因子。

- ❑ 文档权重 (document boost): 索引期赋予某个文档的权重值。
- ❑ 字段权重 (field boost): 查询期赋予某个字段的权重值。
- ❑ 协调因子 (coord): 基于文档中词项个数的协调因子，一个文档命中了查询中的词项越多，得分越高。
- ❑ 逆文档频率 (inverse document frequency): 一个基于词项的因子，用来告诉评分公式该词项有多么罕见。逆文档频率越高，词项就越罕见。评分公式利用该因子，为包含罕见词项的文档加权。
- ❑ 长度范数 (Length norm): 每字段的基于词项个数的归一化因子（在索引期被计算并存储在索引中）。一个字段包含的词项数越多，该因子的权重越低，这意味着 Apache Lucene 评分公式更“喜欢”包含更少词项的字段。
- ❑ 词频 (Term frequency): 一个基于词项的因子，用来表示一个词项在某个文档中出现了多少次。词频越高，文档得分越高。
- ❑ 查询范数 (Query norm): 一个基于查询的归一化因子，它等于查询中词项的权重平方和。查询范数使不同查询的得分能互相比，尽管这种比较通常是困难和不可行的。

2.1.2 TF/IDF 评分公式

从 Lucene 4.0 版本起，Lucene 引入了多种不同的打分公式，这一点或许你已经有所了解了。不过，我们还是希望在此探索一下默认的 TF/IDF 打分公式的一些细节。请记住，为了调节查询相关性，你并不需要深入理解这个公式的来龙去脉，但是了解它的工作原理却

非常重要，因为这有助于简化相关度调优过程。

1. Lucene 的理论评分公式

TF/IDF 公式的理论形式如下：

$$\text{score}(q, d) = \text{coord}(q, d) * \text{queryBoost}(q) * \frac{V(q) * V(d)}{|V(q)|} * \text{lengthNorm}(d) * \text{docBoost}(d)$$

上面的公式融合了布尔检索模型和向量空间检索模型。我们打算在此讨论理论评分公式，而是直接跳到实践中使用的评分公式，看看 Lucene 内部是如何实现和使用评分公式的。



注意 关于布尔检索模型和向量空间检索模型的知识远远超出了本书的讨论范围，想了解更多相关知识，请参考 http://en.wikipedia.org/wiki/Standard_Boolean_model 和 http://en.wikipedia.org/wiki/Vector_Space_Model。

2. Lucene 的实际评分公式

现在让我们看看 Lucene 实际使用的评分公式：

$$\text{score}(q, d) = \text{coord}(q, d) * \text{queryNorm}(q) * \sum_{t \in q} (\text{tf}(t \text{ in } d) * \text{idf}(t)^2 * \text{boost}(t) * \text{norm}(t, d))$$

也许你已经看到了，评分公式是一个关于查询 q 和文档 d 的函数，正如我们之前提到的一样。有两个因子并不直接依赖查询词项，它们是 coord 和 queryNorm ，这两个因子与查询词项的一个求和公式相乘。

求和公式中每个加数由以下因子连乘所得：词频，逆文档频率，词项权重，范数。范数就是之前我们提到过的长度范数。

这个公式听起来很复杂。请别担心，你并不用记住所有的细节，你只需要意识到哪些因素是与评分有关的即可。从前面的公式我们可以导出一些基本的规则：

- ❑ 越罕见的词项被匹配上，文档得分越高。Lucene 认为包含独特单词的文档比包含常见单词的文档更重要。
- ❑ 文档字段越短（包含更少的词项），文档得分越高。通常，Lucene 更加重视较短的文档，因为这些短文档更有可能和我们查询的主题高度吻合。
- ❑ 权重越高（不论是索引期或是查询期赋予的权重值），文档得分越高。因为更高的权重意味着特定数据（文档、词项、短语等）具有更高的重要性。

正如你所见，Lucene 将最高得分赋予同时满足以下条件的文档：包含多个罕见查询词项，词项所在字段较短（该字段索引了较少的词项）。该公式更“喜欢”包含罕见词项的文档。



注意 如果你想了解更多关于 Apache Lucene TF/IDF 评分公式的信息，请参考 Apache lucene 中 TFIDFSimilarity 类的文档：http://lucene.apache.org/core/4_9_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html.

2.1.3 Elasticsearch 如何看评分

总而言之，Elasticsearch 使用了 Lucene 的评分功能，幸运的是 Elasticsearch 允许我们挑选可用的 similarity 类实现，或者自定义 similarity 类，来替换默认的评分算法。不过请记住，Elasticsearch 不仅仅是 Lucene 的简单封装，因为它虽然使用了 Lucene 的评分功能，但不仅限于 Lucene 的评分功能。

用户可以使用各种不同的查询类型，以精确控制文档评分的计算。例如使用 function_score 查询时，可以通过使用脚本 (scripting) 来改变文档得分，也可以使用 Elasticsearch 0.90 中出现的二次评分功能，通过在返回文档集之上执行另外一个查询，重新计算 top-N 文档的得分。



注意 想了解更多 Apache Lucene 查询类型，请参考 http://lucene.apache.org/core/4_9_0/queries/org/apache/lucene/queries/package-summary.html 上的相关文档。

2.1.4 一个例子

现在，我们已经了解评分的工作原理。接下来我们看一个在现实生活中应用评分的简单例子。首先我们需要创建一个名为 scoring 的新索引。使用如下命令创建这个索引：

```
curl -XPUT 'localhost:9200/scoring' -d '{
  "settings" : {
    "index" : {
      "number_of_shards" : 1,
      "number_of_replicas" : 0
    }
  }
}'
```

简单起见，我们使用了只有一个物理分片和 0 个副本的索引（我们不需要在这个例子中关心分布式文档频率）。我们需要索引一个简单的文档，代码如下：

```
curl -XPOST 'localhost:9200/scoring/doc/1' -d '{"name": "first document"}'
```

接着我们执行一个简单的匹配 (match) 查询，查询的词条是 “document”。

```
curl -XGET 'localhost:9200/scoring/_search?pretty' -d '{
  "query" : {
    "match" : { "name" : "document" }
  }
}'
```

Elasticsearch 返回的结果如下:

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.19178301,
    "hits" : [ {
      "_index" : "scoring",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 0.19178301,
      "_source": {"name": "first document"}
    } ]
  }
}
```

显然, 刚才索引的这个文档被匹配上了, 并且被赋予了得分。我们可以通过下面这条命令来查看得分的计算过程:

```
curl -XGET 'localhost:9200/scoring/doc/1/_explain?pretty' -d '{
  "query" : {
    "match" : { "name" : "document" }
  }
}'
```

Elasticsearch 返回的结果如下:

```
{
  "_index" : "scoring",
  "_type" : "doc",
  "_id" : "1",
  "matched" : true,
  "explanation" : {
    "value" : 0.19178301,
    "description" : "weight(name:document in 0)
    [PerFieldSimilarity], result of:",
    "details" : [ {
      "value" : 0.19178301,
```



```

    "description" : "fieldWeight in 0, product of:",
    "details" : [ {
      "value" : 1.0,
      "description" : "tf(freq=1.0), with freq of:",
      "details" : [ {
        "value" : 1.0,
        "description" : "termFreq=1.0"
      } ]
    }, {
      "value" : 0.30685282,
      "description" : "idf(docFreq=1, maxDocs=1)"
    }, {
      "value" : 0.625,
      "description" : "fieldNorm(doc=0)"
    } ]
  } ]
}

```

可以看出, Elasticsearch 给出了针对给定文档和查询的详细的得分计算过程。同时可以看出, 得分等于词项频率 (本例中是 1) 和逆文档频率 (0.30685282) 以及字段范数 (0.625) 的乘积。

现在, 我们再把另一个文档加入索引。

```

curl -XPOST 'localhost:9200/scoring/doc/2' -d '{"name":"second
example document"}'

```

此时, 如果执行最开始的查询, 我们将看到如下响应:

```

{
  "took" : 6,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "hits" : {
    "total" : 2,
    "max_score" : 0.37158427,
    "hits" : [ {
      "_index" : "scoring",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 0.37158427,
      "_source": {"name": "first document"}
    }, {
      "_index" : "scoring",
      "_type" : "doc",
      "_id" : "2",
      "_score" : 0.2972674,
      "_source": {"name": "second example document"}
    }
  ]
}

```

```
    } }
  }
}
```

现在，可以对比一下 TF/IDF 评分公式在现实场景中的工作了。在把第 2 个文档索引到相同分片后（请记住我们创建的索引只有一个分片且没有副本），得分发生了变化，尽管此时的查询和刚才的一样。这是因为一些影响得分的因子已经改变了。比如，逆文档频率变了，因此得分也会跟着改变。我们还需要注意对比一下两个文档的得分。我们查询了一个单词“document”，查询匹配上了两个文档的相同字段的相同词项。第 2 个文档的得分为什么较低，是因为和第 1 个文档相比，它的 name 字段多了一个词项。根据先前的知识储备，我们知道，文档越短，Lucene 给出的得分越高。

希望这个简短的介绍会让你对评分工作机制认识得更清楚，在你需要优化查询时理解目标查询的工作过程。

2.2 查询改写

之前我们探讨了评分机制，这些知识非常珍贵，特别是当你尝试改进查询相关性时。我们还认为，在对查询进行调试时，也很有必要搞清楚查询是如何执行的。因此我们决定在本节介绍一下查询改写是如何工作的，为什么需要查询改写，以及我们应该如何控制它。

如果你之前使用过诸如前缀查询或通配符查询之类的查询类型，那么你会了解这些都是基于多词项的查询，它们都涉及查询改写。Elasticsearch 使用查询改写是出于对性能的考虑。从 Lucene 的角度来看，所谓的查询改写操作，就是把费时的原始查询类型实例改写成一组性能更高的查询类型实例，从而加快查询执行速度。查询改写过程对客户端不可见，不过最好能够知道我们可以修改查询改写过程。举个例子，让我们看看 Elasticsearch 是如何处理前缀查询的。

2.2.1 前缀查询示例

演示查询改写过程的最好方式莫过于通过范例深入了解该过程的内部实现机制，尤其是要去了解原始查询中的词项是如何被改写成目标查询中那些词项的。假设我们索引了下面这些文档中的数据：

```
curl -XPUT 'localhost:9200/clients/client/1' -d '{
  "id": "1", "name": "Joe"
}'
curl -XPUT 'localhost:9200/clients/client/2' -d '{
  "id": "2", "name": "Jane"
}'
```

```
curl -XPUT 'localhost:9200/clients/client/3' -d '{
  "id": "3", "name": "Jack"
}'
curl -XPUT 'localhost:9200/clients/client/4' -d '{
  "id": "4", "name": "Rob"
}'
```

现在我们想找出索引中所有 name 字段以字母 j 开头的文档。简单起见，我们在 clients 索引中执行以下查询：

```
curl -XGET 'localhost:9200/clients/_search?pretty' -d
{
  "query" : {
    "prefix" : {
      "name" : {
        "prefix" : "j",
        "rewrite" : "constant_score_boolean"
      }
    }
  }
}
```

这里使用了一个简单的前缀查询，想检索出所有 name 字段以字母 j 开头的文档。我们同时也设置了查询改写属性以确定执行查询改写的具体方法，不过现在我们跳过该参数，具体的参数值将在本章的后续部分讨论。

执行前面的查询以后，我们将得到下面的结果：

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 3,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "clients",
      "_type" : "client",
      "_id" : "3",
      "_score" : 1.0,
      "_source":{
        "id": "3", "name": "Jack"
      }
    }, {
      "_index" : "clients",
      "_type" : "client",
```



```

      "_id" : "2",
      "_score" : 1.0,
      "_source":{
        "id":"2", "name":"Jane"
      }
    }, {
      "_index" : "clients",
      "_type" : "client",
      "_id" : "1",
      "_score" : 1.0,
      "_source":{
        "id":"1", "name":"Joe"
      }
    }
  ]
}

```

如你所见，返回结果中有 3 个文档，这些文档的 `name` 字段以字母 `j` 开头。我们并没有显式设置待查询索引的映射，因此 Elasticsearch 探测出了 `name` 字段的映射，并将其设置为字符串类型并进行文本分析。可使用下面的命令进行检查：

```
curl -XGET 'localhost:9200/clients/client/_mapping?pretty'
```

Elasticsearch 将返回类似下面的结果：

```

{
  "client" : {
    "properties" : {
      "id" : {
        "type" : "string"
      },
      "name" : {
        "type" : "string"
      }
    }
  }
}

```

2.2.2 回到 Apache Lucene

现在我们回到 Lucene。如果你还记得 Lucene 倒排索引是如何构建的，你会指出倒排索引中包含了词项、词频以及文档指针（如果忘了，请重新阅读 1.1 节）。现在我们看看之前存储到 `clients` 索引中的数据大概是如何组织的。

Term	Count	Docs
jack	1	<3>
jane	1	<2>
joe	1	<1>
rob	1	<4>

Term 这一列非常重要。如果我们去探究 Elasticsearch 和 Lucene 的内部实现，将会发现前缀查询被改写为下面这种查询：

```
ConstantScore(name:jack name:jane name:joe)
```

我们可以用 Elasticsearch API 来检查重写片段。首先，使用 Explain API 执行如下命令：

```
curl -XGET 'localhost:9200/clients/client/1/_explain?pretty' -d '{
  "query" : {
    "prefix" : {
      "name" : {
        "prefix" : "j",
        "rewrite" : "constant_score_boolean"
      }
    }
  }
}'
```

执行结果如下：

```
{
  "_index" : "clients",
  "_type" : "client",
  "_id" : "1",
  "matched" : true,
  "explanation" : {
    "value" : 1.0,
    "description" : "ConstantScore(name:joe), product of:",
    "details" : [ {
      "value" : 1.0,
      "description" : "boost"
    }, {
      "value" : 1.0,
      "description" : "queryNorm"
    } ]
  }
}
```

可以看到，Elasticsearch 对 name 字段使用了一个词项是 joe 的 constant_score 查询。当然，这一步发生在 Lucene 中，Elasticsearch 实际上只是从缓存中获取这些词项。这一点可以用 Validate 查询 API 来验证。

```
curl -XGET 'localhost:9200/clients/client/_validate/query?explain&pretty'
-d '{
  "query" : {
    "prefix" : {
      "name" : {
        "prefix" : "j",
        "rewrite" : "constant_score_boolean"
      }
    }
  }
}'
```

```

    }
  }
}
}'

```

Elasticsearch 返回的结果如下：

```

{
  "valid" : true,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "explanations" : [ {
    "index" : "clients",
    "valid" : true,
    "explanation" : "filtered(name:j*)->cache(_type:client)"
  } ]
}

```

2.2.3 查询改写的属性

当然，多词项查询的 `rewrite` 属性也可以支持除了 “`constant_score_boolean`” 之外的其他取值。我们可以通过这个属性来控制查询在 Lucene 内部的改写方式。我们可以将 `rewrite` 参数存放在代表实际查询的 JSON 对象中，例如，像下面的代码这样：

```

{
  "query" : {
    "prefix" : {
      "name" : "j",
      "rewrite" : "constant_score_boolean"
    }
  }
}

```

现在让我们来看看 `rewrite` 参数有哪些选项可以配置。

- ❑ `scoring_boolean`：该选项将每个生成的词项转化为布尔查询中的一个或从句（Boolean should clause）。这种改写方法需要针对每个文档都计算得分。因此，这种方法比较耗费 CPU（因为要计算和保存每个词项的得分），而且有些查询生成了太多的词项，以至于超出了布尔查询默认的 1024 个从句的限制。默认的布尔查询限制可以通过设置 Elasticsearch.yml 文件的 `index.query.bool.max_clause_count` 属性来修改。用户需谨记，改写后的布尔查询的从句数越多，查询性能越低。
- ❑ `constant_score_boolean`：该选项与前面提到过的 `scoring_boolean` 类似，但是 CPU 耗费更少，这是因为并不计算每个从句的得分，而是每个从句得到一个与查询权重相

同的一个常数得分，默认情况下等于 1，我们也可以通过设置查询权重来改变这个默认值。与 `scoring_boolean` 类似，该选项也有布尔从句数的限制。

- ❑ `constant_score_filter`：正如 Lucene 的 Javadocs 描述的那样，该选项按如下方式改写原始查询——通过顺序遍历每个词项来创建一个私有的过滤器，标记所有包含这个词项的文档。命中的文档被赋予一个与查询权重相同的常量得分。当命中词项数或文档数较大时，该方法比 `scoring_boolean` 和 `constant_score_boolean` 执行速度更快。
- ❑ `top_terms_N`：该选项将每个生成的词项转化为布尔查询中的一个或从句，并保存计算出来的查询得分。与 `scoring_boolean` 不同之处在于，该方法只保留最佳的 `N` 个词项，以避免触及布尔从句数的限制，并提升查询整体性能。
- ❑ `top_terms_boost_N`：该选项与 `top_terms_N` 类似，不同之处在于它的文档得分不是通过计算得出的，而是被设置为跟查询权重（boost）一致，默认值为 1。



注意

当 `rewrite` 属性设置为 `constant_score_auto` 或者没有设置时，Elasticsearch 会根据查询的类型及其构造方式来决定是使用 `constant_score_filter` 还是 `constant_score_boolean`。

现在，让我们再看一个例子。如果我们想在范例查询中使用 `top_terms_N` 选项，并且 `N` 的值设置为 2，那么查询看起来与下面的代码类似：

```
{
  "query" : {
    "prefix" : {
      "name" : {
        "prefix" : "j",
        "rewrite" : "top_terms_2"
      }
    }
  }
}
```

从 Elasticsearch 返回的结果中可以看出，和我们之前使用的查询不同，这里的文档得分都不等于 1.0。

```
{
  "took" : 3,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 3,
    "max_score" : 0.30685282,
    "hits" : [ {
```

```
{
  "_index" : "clients",
  "_type" : "client",
  "_id" : "3",
  "_score" : 0.30685282,
  "_source":{
    "id":"3", "name":"Jack"
  }
}, {
  "_index" : "clients",
  "_type" : "client",
  "_id" : "2",
  "_score" : 0.30685282,
  "_source":{
    "id":"2", "name":"Jane"
  }
}, {
  "_index" : "clients",
  "_type" : "client",
  "_id" : "1",
  "_score" : 0.30685282,
  "_source":{
    "id":"1", "name":"Joe"
  }
}
} ]
}
```

这是因为 `top_terms_N` 需要保留得分最高的 N 个词项。

结束本节之前，读者应该会产生一个疑问，我们如何决定何时采用何种查询改写方法？该问题的答案更多地取决于您的应用场景。简单来说，如果您能接受较低的精度和相关性（但是追求更高的性能），那么可以采用 `top-N` 查询改写方法。如果您需要更高的查询精度和更好的相关性（同时可以接受较低的性能），那么应该采用布尔方法。

2.3 查询模板

在应用程序迭代的同时，它的运行环境很可能会越来越复杂。在你所处的组织中，很可能同一个应用程序的不同部分分别有专人负责，比如说，至少有一个前端工程师和一个负责数据库层的后端工程师。将应用程序划分为几个模块分别开发的方式非常便捷高效，它能够让开发人员针对程序的不同部分并行进行开发工作，而无需在开发者之间和开发小组内部时刻同步代码。当然，你正在阅读的这本书不是关于项目管理的，而是聚焦于搜索的，因此让我们回到正题上。有时候，我们可以整理出程序使用的所有查询语句交给搜索引擎工程师，让他们协助从性能和相关性两个方面对查询语句进行优化。这种做法通常是很有帮助的。在这种情况下，应用程序开发者只需要把查询传递给 Elasticsearch，而不需要

考虑查询语句的构造、查询 DSL 语法、查询结果过滤等细节知识。

2.3.1 引入查询模板

自 Elasticsearch 1.1.0 版本开始，我们可以自定义查询模板。让我们回到本书开头的在线书店例子中。假定我们已经确定了需要传递给 Elasticsearch 的查询语句的类型，不过查询结构并未最终确定，我们还需要对它进行微调和优化。通过使用查询模板，我们可以快速构建出查询的基础骨架，然后让应用程序来提供对应的参数，最终由 Elasticsearch 完成查询参数的替换。

假定我们有一个针对 library 索引的查询语句，可以返回最相关的书籍记录。在这个查询中，我们还允许用户选择是否对书籍的库存状态做筛选。在这个场景中，我们需要传入两个参数——一个查询短语和一个代表书籍库存状态的布尔变量。最初的简化示例如下：

```
{
  "query": {
    "filtered": {
      "query": {
        "match": {
          "_all": "QUERY"
        }
      },
      "filter": {
        "term": {
          "available": BOOLEAN
        }
      }
    }
  }
}
```

代码中的 QUERY 和 BOOLEAN 是占位符，代表应用程序传递给查询的变量。显然这个查询语句对当前示例场景来说实在太简陋了，不过之前我们已经说过，这只是它的最初版本，我们马上将对它进行改进。

既然已经有了最初版本的查询语句，我们可以基于它创建第一个查询模板。对该查询语句做简单修改如下：

```
{
  "template": {
    "query": {
      "filtered": {
        "query": {
          "match": {
            "_all": "{{phrase}}"
          }
        }
      }
    },
  },
}
```



```

    "filter": {
      "term": {
        "available": "{{avail}}"
      }
    },
  },
  "params": {
    "phrase": "front",
    "avail": true
  }
}

```

可以看出，原来的占位符被替换成了 `{{phrase}}` 和 `{{avail}}` 两个变量，并且添加了一个新的 `params` 片段。当 Elasticsearch 在解析查询语句时，遇到一个 `{{phrase}}` 变量，它将尝试从 `params` 片段中查找出名为 `phrase` 的参数，并用参数值替换掉 `{{phrase}}` 变量。通常，我们需要把参数值放到 `params` 片段中，并在 `query` 中使用形如 `{{var}}` 的标记来引用 `params` 片段中参数名为 `var` 的参数。此外，查询本身被嵌套进一个 `template` 元素中。通过这种方式，我们实现了查询的参数化。

接下来让我们使用 HTTP GET 请求把以下查询语句发送给地址为 `/library/_search/template` 的 REST 端点（注意这里不是我们通常使用的 `/library/_search` 端点）。请求命令构造如下：

```

curl -XGET 'localhost:9200/library/_search/template?pretty' -d '{
  "template": {
    "query": {
      "filtered": {
        "query": {
          "match": {
            "_all": "{{phrase}}"
          }
        },
      },
      "filter": {
        "term": {
          "available": "{{avail}}"
        }
      }
    },
  },
  "params": {
    "phrase": "front",
    "avail": true
  }
}'

```

字符串形式的查询模板

查询模板也可以以字符串的形式提供。比如，刚才的查询模板可以变成这样：

```
{
  "template": "{ \"query\": { \"filtered\": { \"query\": {
    \"match\": { \"_all\": \"{phrase}\" } }, \"filter\": {
    \"term\": { \"available\": \"{avail}\" } } } } }",
  "params": {
    "phrase": "front",
    "avail": true
  }
}
```

可见，这种形式不太适合阅读和书写，每个引号都需要被转义，换行符容易引发格式问题，因此需要避免使用。尽管如此，如果你需要使用 Mustache（一个模板引擎，我们将在下一小节探讨），则必须使用这种格式（至少在 Elasticsearch 的 1.1.0 到 1.4.0 之间的所有版本中必须这样做）。



注意

本书写作时，笔者所使用的 Elasticsearch 相关版本中有一个关于查询模板的小陷阱。如果你提供的查询模板中有错误，被 Elasticsearch 检测到后，会把错误写到服务日志里，但是从 API 的视角来看，错误查询将被忽略，接口将返回所有文档，就好像你刚刚发送了一个 `match_all` 查询一样。记得复查你的查询模板，直到这个缺陷不再存在。

2.3.2 Mustache 模板引擎

Elasticsearch 使用 Mustache 模板引擎（参考 <http://mustache.github.io>）来为查询模板生成可用的查询语句。如你所见，每个变量被双大括号包裹，这一点是 Mustache 规范要求的，是该模板引擎间接引用变量的方式。Mustache 模板引擎的完整语法不在本书讨论范围内，不过我们可以在这里简单介绍一下它最具魅力的部分，包括条件表达式、循环和默认值。



注意

Mustache 语法的详细内容请参阅 <http://mustache.github.io/mustache.5.html>。

1. 条件表达式

`{{val}}` 表达式用来插入变量 `val` 的值。`{{#val}}` 和 `{{/val}}` 则用来在变量 `val` 取值计算为 `true` 时把位于它们之间的变量标记替换为变量值。

我们看一下下面这个示例：

```
curl -XGET 'localhost:9200/library/_search/template?pretty' -d '{
  "template": "{ {{#limit}}\"size\": 2 {{/limit}}}",
  "params": {
    "limit": false
  }
}'
```

这个命令将返回 library 索引中的所有文档。不过，假如我们把 limit 参数的取值改为 true，则再次查询后，我们将只能得到两个文档。这是因为判断条件满足了，模板内容因此被激活。



注意

不幸的是，似乎直到本书写作时，笔者所使用的 Elasticsearch 版本在处理条件表达式时仍然有些问题。比如，其中一个相关问题可以在这里看到：<https://github.com/Elasticsearch/Elasticsearch/issues/8308>。我们决定保留条件表达式这一小节，以期望相关问题都能在未来得到解决。使用条件表达式可以更方便地构造查询模板。

2. 循环

循环结构定义和条件表达式一模一样，都位于 {{#val}} 和 {{/val}} 之间。如果表达式中变量取值是数组，则可以使用 {{.}} 标记来指代当前变量值。

例如，假定我们需要模板引擎遍历一个词项数组来生成一个词项查询，可以执行如下命令：

```
curl -XGET 'localhost:9200/library/_search/template?pretty' -d '{
  "template": {
    "query": {
      "terms": {
        "title": [
          "{{#title}}",
          "{{.}}",
          "{{/title}}"
        ]
      }
    }
  },
  "params": {
    "title": [ "front", "crime" ]
  }
}'
```


3. 默认值

默认值标记允许我们在参数未定义时给它设置默认取值。比如，给 var 变量设置默认值语法的代码如下：

```
{{var}}{{^var}}default value{{/var}}
```

举个例子，假定我们要给查询模板中的 phrase 参数设置默认值 “crime”，可以使用如下命令：

```
curl -XGET 'localhost:9200/library/_search/template?pretty' -d '{
  "template": {
    "query": {
      "term": {
        "title": "{{phrase}}{{^phrase}}crime{{/phrase}}"
      }
    },
    "params": {
      "phrase": "front"
    }
  }
}'
```

这个命令将从 Elasticsearch 查询出所有 title 字段中包含 front 的文档。而如果我们在 params 片段中不指定 phrase 参数的值，则使用 crime 来代替。

2.3.3 把查询模板保存到文件

抛开之前定义模板的方式不说，我们距离把查询跟应用程序解耦还有相当长的一段路要走。我们能够做的仅仅是把查询语句参数化，而整个查询模板字符串仍然需要保存在应用程序中。幸运的是，有一种简单的方法来改变目前这种查询定义方式，它允许 Elasticsearch 从 config/scripts 目录中动态读取查询模板。

举例来说，让我们创建一个名为 bookList.mustache 的文件（在 config/scripts 目录中）。使用如下命令：

```
{
  "query": {
    "filtered": {
      "query": {
        "match": {
          "_all": "{{phrase}}"
        }
      },
      "filter": {
        "term": {
          "available": "{{avail}}"
        }
      }
    }
  }
}
```

```

    }
  }
}
}
}

```

接下来我们就可以在查询中用模板名称来使用该文件的内容了（模板名称就是模板文件名称去掉 `.mustache` 后缀）。例如，如果我们使用 `bookList` 模板，则可以使用如下命令：

```

curl -XGET 'localhost:9200/library/_search/template?pretty' -d '{
  "template": "bookList",
  "params": {
    "phrase": "front",
    "avail": true
  }
}'

```



注意

Elasticsearch 有一个非常方便的特性：它可以无需重启就检测到模板文件的变更。当然，我们还是需要在每个负责查询的 Elasticsearch 节点上部署查询模板文件。从 Elasticsearch 1.4.0 版本开始，你可以把模板索引到一个名为 `.scripts` 的特殊索引中。更多相关信息请参考 Elasticsearch 的官方文档：<http://www.Elasticsearch.org/guide/en/Elasticsearch/reference/current/search-template.html>。

2.4 过滤器的使用及作用原理

接下来，我们一起认识一下 Elasticsearch 提供的过滤功能。初看起来，过滤好像一个多余的功能，因为几乎每个过滤器在 Elasticsearch 查询 DSL 中都以一个与查询代码极其相似的方式呈现的。不过，这些过滤器一定有其独到之处，不然它们就不会在查询性能优化时被广泛使用和推荐了。本节我们将着重探讨为什么过滤器如此重要，它们的工作原理，以及 Elasticsearch 都提供了哪些过滤器给我们使用。

2.4.1 过滤及查询相关性

普通查询和过滤的第一个差异在于它们对文档打分的影响。让我们举例对比一下查询和过滤的输出。首先执行如下查询：

```

curl -XGET "http://127.0.0.1:9200/library/_search?pretty" -d'
{
  "query": {

```

```

      "term": {
        "title": {
          "value": "front"
        }
      }
    }
  }
}'

```

这个查询的结果如下：

```

{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.11506981,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "1",
      "_score" : 0.11506981,
      "_source":{ "title": "All Quiet on the Western
Front", "otitle": "Im Westen nichts Neues", "author": "Erich
Maria Remarque", "year": 1929, "characters": ["Paul Bäumer",
"Albert Kropp", "Haie Westhus", "Fredrich Müller",
"Stanislaus Katczinsky", "Tjaden"], "tags":
["novel"], "copies": 1,
"available": true, "section" : 3}
    } ]
  }
}

```

这个查询没有任何特异之处。Elasticsearch 将返回所有在 title 字段中包含“front”的文档。需要指出的是，每个和查询匹配的文档都会被计算得分，其中得分最高的一组文档被作为查询结果返回给用户。在本例中，该查询返回了一篇得分为 0.11506981 的文档。以上这些就是查询的一般行为。

接着我们来对比一下查询和过滤。在一个同时包含查询和过滤的例子中，我们将加入一段代码片段，限制返回文档只能有一个副本（copies 字段取值为 1）。不使用过滤的查询方式如下：

```

curl -XGET "http://127.0.0.1:9200/library/_search?pretty" -d'
{
  "query": {

```



```

    "bool": {
      "must": [
        {
          "term": {
            "title": {
              "value": "front"
            }
          }
        },
        {
          "term": {
            "copies": {
              "value": "1"
            }
          }
        }
      ]
    }
  }
}

```

Elasticsearch 返回的查询结果和上一个查询非常相似:

```

{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.98976034,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "1",
      "_score" : 0.98976034,
      "_source":{ "title": "All Quiet on the Western
Front","otitle": "Im Westen nichts Neues","author": "Erich
Maria Remarque","year": 1929,"characters": ["Paul Bäumer",
"Albert Kropp", "Haie Westhus", "Fredrich Müller",
"Stanislaus Katczinsky", "Tjaden"],"tags":
["novel"],"copies": 1,
"available": true, "section" : 3}
    } ]
  }
}

```

上面这段查询代码中的 bool 查询由两个 term 查询构成，每个结果文档都需要同时匹配这两个 term 查询。这个查询返回了和上一查询相同的文档，不过文档得分变成了 0.98976034。这和我们读完 2.1 节后的期望一致：每个词项都会影响得分。

接下来我们来看看使用过滤的查询方式，在 title 字段匹配 “front” 的查询，同时针对 copies 字段进行过滤。

```
curl -XGET "http://127.0.0.1:9200/library/_search?pretty" -d'
{
  "query": {
    "term": {
      "title": {
        "value": "front"
      }
    }
  },
  "post_filter": {
    "term": {
      "copies": {
        "value": "1"
      }
    }
  }
}'
```

现在，我们构造好了一个 term 查询，同时还添加了一个 term 过滤器。从下面的返回代码中可以看出，输出的文档和不使用过滤时一样，不过文档得分发生了变化。

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.11506981,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "1",
      "_score" : 0.11506981,
      "_source":{ "title": "All Quiet on the Western
Front","otitle": "Im Westen nichts Neues","author": "Erich
```

```
    "Maria Remarque", "year": 1929, "characters": ["Paul Bäumer",  
    "Albert Kropp", "Haie Westhus", "Fredrich Müller",  
    "Stanislaus Katschinsky", "Tjaden"], "tags":  
    ["novel"], "copies": 1,  
    "available": true, "section" : 3}  
  }  
}
```

这个文档的得分为 0.11506981，这和本节最开始的查询结果一模一样。通过得分对比我们得出结论：过滤不影响文档得分。



注意 旧版 Elasticsearch 使用 “filter” 而不是上述代码中的 “post_filter” 来标识查询语句中的过滤片段。在 1.x 版本中，这两种标记方式都可以正常使用，不过请注意，“filter” 方式可能将在之后的版本中停用。

一般来说，查询和过滤在工作过程中存在一个主要的差异。过滤的唯一目的是用特定筛选条件来缩小结果范围。而查询不仅缩小结果范围，还会影响文档的得分，这一点在强调文档相关性时非常重要，不过需要付出一定的代价：需要额外的 CPU 消耗来计算文档得分。当然，这不是查询和过滤的唯一区别。本节剩余部分将着重探讨过滤器的工作原理和 Elasticsearch 提供的不同过滤方式之间的异同。

2.4.2 过滤器的工作原理

前一小节我们已经提到，过滤不影响所匹配文档的得分。基于两个原因，这一点非常重要。第 1 个原因是性能。针对索引中的一组文档进行过滤操作是非常简单高效的。过滤器持有的关于文档的唯一重要信息是该文档是否匹配这个过滤器——仅仅一个标记而已。

过滤器通过返回一个被称为 DocIdSet (org.apache.lucene.search.DocIdSet) 的数据结构来提供这类匹配信息。这个数据结构的用途是为索引段提供经过过滤器过滤后的数据。它可以使用 Bits 接口 (org.apache.lucene.util.Bits) 的有关实现。Bits 接口可以随机访问过滤器中的文档信息（主要是检查索引段中的某个文档是否和该过滤器匹配）。Bits 的数据结构非常高效，因为 CPU 可以使用位运算来完成过滤（有一个精巧的 CPU 部件用来处理这类运算，详情参考环形移位的维基百科 http://en.wikipedia.org/wiki/Circular_shift）。DocIdSet 还针对内部文档标识的有序集合提供了一个 DocIdSetIterator 迭代器给我们使用。

下表展示了这些类是如何使用 Bits 进行工作的。

doc	bits.get(doc)	Result
1	FALSE	
2	FALSE	
3	TRUE	3
4	TRUE	4

Lucene（以及 Elasticsearch）提供了 DocIdSet 的多种实现来应对不同场景。不同实现的性能各不相同。不过，选择合适的实现是 Lucene 和 Elasticsearch 的职责，我们一般不需要关心这一点，除非我们要针对它们进行功能扩展。



注意

不是所有的过滤器都使用 Bits 结构，比如数值区间过滤器、脚本过滤器、以及基于地理位置的一组过滤器。这些特殊的过滤器选择把数据记录在字段缓存里，然后再遍历所需处理的文档集合，逐个进行过滤操作。这意味着过滤器链条中的下一个过滤器只能获取到匹配前一个过滤器的文档集合。因此，可以针对这些过滤器进行优化，比如把最重的（译者注：匹配文档最多的，或者性能最差的）过滤器放到过滤器链的最后去执行。

布尔过滤器和与或非过滤器

我们在《Elasticsearch Server, Second Edition》一书中探讨了过滤器的有关知识，在这里只需要提醒读者注意一点：与或非过滤器不使用 Bits，而布尔过滤器使用了 Bits。因此，请尽可能使用布尔过滤器。与或非过滤器一般在需要脚本过滤、地理位置过滤和数值区间过滤时使用。还需要注意的是，如果把任何不使用 Bits 的过滤器嵌套在与或非过滤器中，它们同样不会用到 Bits。

一般来讲，在组合使用多个处理器时，如果其中包含不使用 Bits 的处理器，则需要使用与或非处理器来对它们进行组合。而如果要组合的所有处理器都使用 Bits，则可以选择使用布尔过滤器来组合它们。

2.4.3 性能考量

通常，过滤器都是很快的。这一点有多种原因。首先，最重要的一点是，由过滤器所处理的查询部分不需要计算文档得分。之前我们就提到过，打分过程与给定查询和索引中的文档集合密切相关。



注意

使用过滤器时需要注意一点：在 Elasticsearch 1.4.0 版本发布后，执行嵌套查询时所使用的 `bitsets` 默认提前就加载好了。这样做的目的是使嵌套查询执行得更快，不过可能伴随内存使用问题。可以通过设置 `index.load_fixed_bitset_filters_eagerly` 配置项为 `false` 来禁用提前加载。如果需要查看固定大小 `bitsets` 的内存占用情况可以执行以下命令：`curl -XGET 'localhost:9200/_cluster/stats?human&pretty'`，在响应中寻找 `fixed_bit_set_memory_in_bytes` 属性即可。

在使用过滤器时，过滤结果不依赖于查询，因此过滤结果可以被轻易地缓存起来供后续查询使用。值得一提的是，每个 Lucene 索引段都有一个过滤结果缓存。这意味着无需在每次 `commit` 时重建缓存，重建操作只发生在段生成和合并时。



注意

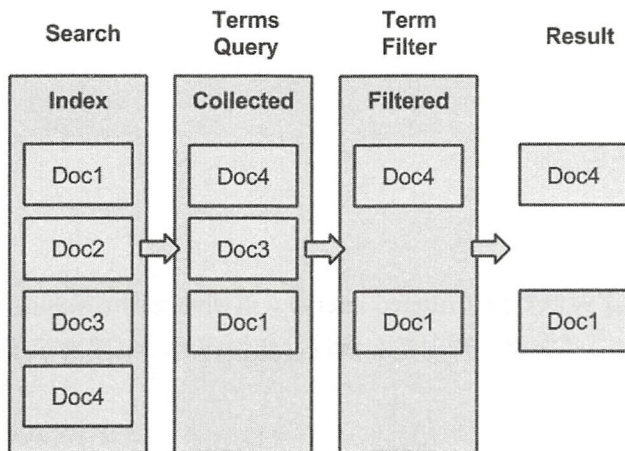
当然，有得必有失，过滤器也有一些不好的地方。不是所有的过滤器都可以被缓存。考虑那些依赖于当前时间的过滤器，对它们做缓存不会有任何意义。在某些场景下不值得做缓存，因为可能存在非常多的唯一值，缓存命中率极低，比如基于地理位置过滤的场景。

2.4.4 后置过滤和过滤查询

如果某人说过滤比实现相同功能的查询执行更快，这不一定是真的。的确，过滤器需要考虑的东西更少，并且可以在后续查询中复用，不过 Lucene 早就针对查询做了高度优化，以确保查询能够高速执行，甚至在考虑文档评分的情境下。当然，如果匹配结果数量极多，过滤器会执行得更快一些。不过，还有一些事我们没有告诉你。某些时候，在使用后置过滤（`post_filter`）时，Elasticsearch 查询的执行速度没有我们期望的那么快。假如我们执行如下查询：

```
curl -XGET 'http://127.0.0.1:9200/library/_search?pretty' -d '{
  "query": {
    "terms": {
      "title": [ "crime", "punishment", "complete", "front" ]
    }
  },
  "post_filter" : {
    "term": {
      "available": {
        "value": true,
        "_cache": true
      }
    }
  }
}'
```

下图展示了查询的执行过程：



当然，针对大量数据的过滤是很有价值的。不过在本例中，我们使用现有的少量数据。从上图可见，索引中包含 4 个文档。例子中的 terms 查询匹配了 3 个文档：Doc1、Doc3 和 Doc4。每个匹配的文档都被计算得分并根据得分做了排序。之后，post_filter 开始工作。在索引的所有文档中，它只通过了两个文档：Doc1 和 Doc4。可以看到，一共传递给过滤器 3 个文档，而只有其中的两个被作为结果输出。既然如此，还有必要对 Doc3 计算得分吗？本例我们浪费了一部分 CPU 时间来计算一个最终不匹配的文档的得分。如果类似的文档数量很多，这将是一个性能问题。



注意 本例中我们使用了 term 过滤器。该过滤器在 Elasticsearch 1.5 版本之前都是默认缓存的。而从 1.5 版本开始，默认不再缓存（参考 <https://github.com/Elasticsearch/Elasticsearch/pull/7583>）。因此，我们在例子中使用 term 过滤器时特意使用了强制缓存。

让我们修改一下这个查询，让文档过滤操作发生在 Scorer 计算文档得分之前。修改后的查询如下：

```
curl -XGET 'http://127.0.0.1:9200/library/_search?pretty' -d '{
  "query": {
    "filtered": {
      "query": {
        "terms": {
          "title": [ "crime", "punishment", "complete", "front" ]
        }
      },

```


离后置过滤。这一规则在绝大多数情况下是正确的，不过在某些条件下，存在例外情况。经验法则告诉我们，开销最大的操作需要移动到查询处理链条的尾部。如果过滤器执行很快，开销很小，并且易于缓存，很简单，直接选择过滤查询即可。相反，如果过滤器执行很慢，CPU 开销大，并且难于缓存（比如有大量唯一值的情况），请使用后置过滤，或者尝试优化过滤器。优化途径包括简化过滤器和使得过滤器对缓存更友好，比如，可以降低时间区间过滤器的时间粒度。

2.5 选择正确的查询方式

在《Elasticsearch Server, Second Edition》一书中，我们详细介绍了 Elasticsearch 的查询 DSL，这一种使用 JSON 结构化的查询语言，可以构建极其复杂的查询语句。不过，在那本书中我们没有探讨在不同的场合可以用到哪些查询方式，以及应该使用哪种查询方式。对于一个在全文搜索引擎领域没有经验储备的人来说，Elasticsearch 提供的查询方式显得太多了，而且容易让人迷惑。因此我们将在本书中对这方面的知识做一些深入探讨，从而引导读者如何选择合适的查询方式。

我们把本节内容划分为独立的两部分。第 1 部分试图对所有查询方式进行分类，并告诉你在特定分类下的查询将会产生什么输出。第 2 部分将针对每种分类举例加以说明，并探讨分类间的不同。请注意，本节接下来的内容不是对 Elasticsearch 的查询 DSL 的完整阐释，如果你需要了解查询 DSL 的基本知识，请参考《Elasticsearch Server, Second Edition》一书，或者查阅 Elasticsearch 的官方文档：<http://www.Elasticsearch.org/guide/en/Elasticsearch/reference/current/query-dsl.html>。

2.5.1 查询方式分类

当然，对查询方式进行分类是一件艰难的任务，我们也不敢打包票说在这里给出的分类列表是唯一正确的。我们甚至可以说，如果你询问其他 Elasticsearch 使用者，他们可能会给出自己的分类方式，或者声称每个查询方式都可以被归入多个类别。有趣的是，他们有可能是对的。我们也曾考虑过多种分类方式存在的情况，不过，最终我们认为，每个查询方式都可以被归入以下列出的一个或多个类别中。

- ❑ 基本查询：这类查询允许针对索引的一部分进行检索，其输入数据既可以分析也可以不做分析。这类查询的一个关键特征是，不支持在其内部再嵌套其他查询。基本查询的一个示例是 term 查询。
- ❑ 组合查询：在这类查询中可以包含其他查询和过滤器，比如 bool 查询和 dismax 查询。
- ❑ 无分析查询：这类查询不分析输入内容，直接将它们原样传递给 Lucene。term 查询

就是这类查询的一员。

- ❑ 全文检索查询：这类查询成员众多。许多查询都支持全文检索、输入内容分析、同时很可能支持可被 Lucene 识别的查询语法。比如 `match` 查询。
- ❑ 模式匹配查询：这类查询都在查询语句中支持各种通配符。比如，前缀查询可以归入此类。
- ❑ 支持相似度操作的查询：这类查询拥有一个共同的特性——支持近似词语的匹配。这类查询的成员如 `fuzzy_like_this`、`more_like_this` 查询等。
- ❑ 支持打分操作的查询：这类查询非常重要，尤其是在和全文搜索查询组合使用的场景下。这个类别包括那些允许在查询时修改打分计算过程的查询方式。在第 3 章介绍的 `function_score` 查询可以归入此类。
- ❑ 位置敏感查询：这类查询允许我们使用索引中存储的词项位置信息。`span_term` 查询就是一个很好的例子。
- ❑ 结构敏感查询：这类查询的工作基于结构数据，如父子文档结构。这个类别的一个例子是 `nested one` 查询。

当然，我们在这里只讨论查询分类，不探讨过滤器的分类。不过，对过滤器来说，你也可以使用相同的分类逻辑。让我们先把过滤器置之脑后。在距离阐释每种查询类别之前，我们先简短描述一下每种查询类别的目的。

1. 基本查询

在基本查询内部不可以包含其他查询，它们只有索引检索这一个用途。这类查询通常作为其他复杂查询的一部分或者单独传递给 Elasticsearch。你可以把基本查询比作修筑大厦的砖块，而大厦就是各种复杂查询。举个例子，如果你想匹配某个文档中的一个特定词项，且没有其他要求，可以考虑使用基本查询。本例中，`match` 查询就能很好地满足需求，无需再跟其他查询组合使用。

归属于基本查询的一些查询方式举例如下：

- ❑ `match` 查询：一种（实际上指好几种）查询方式，适用于执行全文检索且需要对输入进行分析的场景。一般来说，当需要分析输入内容却不需要完整 Lucene 查询语法支持时，可以使用这种查询方式。这种查询不需要进行查询语法解析，发生解析错误的概率极低，因此特别适合接收用户输入文本的场景。
- ❑ `match_all` 查询：这个查询匹配所有文档，常用于需要对所有索引内容进行归类处理的场景。
- ❑ `term` 查询：一种简单的、无需对输入进行分析的查询方式，可以查询单个词语。这种查询方式的使用场景包括针对不分词字段进行检索，比如在我们的测试代码中检

索 tags 字段。term 查询还经常跟过滤器配合使用，比如在我们的测试代码中针对 category 字段进行过滤操作。

简单查询分类可包括：match, multi_match, common, fuzzy_like_this, fuzzy_like_this_field, geoshape, ids, match_all, query_string, simple_query_string, range, prefix, regexp, span_term, term, terms, wildcard 查询。

2. 组合查询

组合查询的唯一用途是把其他查询组合在一起使用。如果说简单查询是建造高楼的砖块，组合查询就是粘合这些砖块的水泥。我们可以把组合查询无穷嵌套，用来构建极其复杂的查询，唯一能够阻止我们这样嵌套的障碍是性能。

组合查询的一些示例和用法如下。

❑ **bool 查询**：最常用的组合查询方式之一。能够把多个查询用布尔逻辑组织在一起，可以控制查询的某个子查询部分是必须匹配、可以匹配还是不应该匹配。如果我们要把匹配不同查询条件的查询组合在一起使用，bool 查询就是一个很好的选择。Bool 查询还可以用在这样的场景：我们希望结果文档的最终得分为所有子查询得分的和。

❑ **dis_max 查询**：一种非常有用的查询方式。这种查询的结果文档得分和最高权重的子查询打分高度相关，而不是如 bool 查询那样对所有子查询得分进行求和。Dis_max 查询返回匹配所有子查询的文档，并通过一个简单公式计算最终得分： $\max(\text{各子查询的得分}) + \text{tie_breaker}^\ominus * (\text{非最高得分子查询的得分之和})$ 。如果你希望最高得分子查询能够在打分过程中起决定作用，dis_max 查询是不二选择。

组合查询类别可包括这些查询方式：bool, boosting, constant_score, dis_max, filtered, function_score, has_child, has_parent, indices, nested, span_first, span_multi, span_first, span_multi, span_near, span_not, span_or, span_term, top_children 查询。

3. 无分析查询

有一类查询不会被分析，而是直接传递给 Lucene 索引。这意味着我们既不需要操心分析过程是否如我们期望的方式执行并生成合适的词项，也不需要针对特定的不分词字段执行查询。如果你把 Elasticsearch 当作 NoSQL 数据库使用，这种查询方式就比较适合你。这类查询精确匹配传入的词语，不会使用语言分析器等工具对词语进行分词和其他处理。

以下示例可帮你理解无分析查询的目的。

❑ **term 查询**：即词项查询。当提及无分析查询时，最常用的无分析查询就是 term 查询。

[⊖] tie_breaker 参数是一个 0 到 1 之间的浮点数，取 0 时意为仅取最高得分子查询的得分，和不使用 tie_breaker 的 dis_max 查询效果相同，取 1 则意味着对所有匹配子查询一视同仁，等效于 bool 查询。——译者注

它可以匹配某个字段中的给定值。比如说，如果你希望匹配一个拥有特定标签（我们示例文档中的 `tags` 字段）的文档，可以使用 `term` 查询。

❑ `prefix` 查询：即前缀查询。另一种无需分析的查询方式。前缀查询常用于自动完成功能，用户输入一段文本，搜索系统返回所有以这个文本开头的文档。需要注意的是，尽管前缀查询没有被分析，Elasticsearch 还是对它进行了重写，以确保它能高速执行。这类查询包括：`common`，`ids`，`prefix`，`span_term`，`term`，`terms`，`wildcard` 查询。

4. 全文检索查询

当你需要构建类似 Google 的查询界面时，可以使用这种查询类别。这类查询会根据索引映射配置对输入进行分析，支持 Lucene 语法和打分计算等功能。一般来说，如果查询的一部分文本来自于用户输入，则可以从全文检索查询类别中选择其一，比如 `query_string`、`match` 或 `simple_query_string` 查询。

全文检索查询类别的示例和用法如下。

❑ `simple_query_string` 查询：该查询方式构建于 Lucene 的 `SimpleQueryParser` 类（参考 http://lucene.apache.org/core/4_9_0/queryparser/org/apache/lucene/queryparser/simple/SimpleQueryParser.html，被设计为解析人类可读的查询串）之上。通常情况下，如果你不希望在遭遇解析错误时直接失败，而是尝试给出用户期望的答案，那么这种查询方式是不错的选择。

属于本类的查询方式包括：`match`，`multi_match`，`query_string`，`simple_query_string` 查询。

5. 模式匹配查询

Elasticsearch 直接或间接提供了一些支持通配符的查询方式，比如通配符查询（`wildcard query`）和前缀查询（`prefix query`）。除此之外，我们还可以使用正则表达式查询（`regexp query`），这种查询能够找出内容中包含给定模式的文档。

我们在之前已经展示过一个前缀查询的示例，因此在这里主要介绍一下正则表达式查询。如果想找出其词项匹配某个固定模式的文档，正则表达式查询是唯一的选择。举个例子，假定你的各种日志存储于 Elasticsearch 中，可以使用正则表达式查询找出所有含有如下词项的日志记录：词项以“`err`”前缀开头、以“`memory`”结尾、中间可以有任意数量的字符。最后，需要注意的是，所有模式匹配查询如果包含可匹配海量词项的表达式，性能代价将十分高昂。

本类查询包括：`prefix`，`regexp`，`wildcard` 查询。

6. 支持相似度操作的查询

我们认为这类查询是一些可以根据给定词项查找近似词项或文档的查询方式的集合。

举例来说,假定我们需要找出包含“crimea”近似词项的文档,可以执行一个 fuzzy 查询。这类查询的另一个用途是提供类似“你是不是想找 XXX”的功能。比如你希望找出文档标题和输入文本相似的文档,可以使用 more_like_this 查询。一般来说,你可以使用本类别下的某个查询来查找包含和给定输入近似词项或字段的文档。

属于这个类别的查询有: fuzzy_like_this, fuzzy_like_this_field, fuzzy, more_like_this, more_like_this_field 查询。

7. 支持打分操作的查询

这是一组用于改善查询精度和相关度的查询方式。这类查询可以通过指定自定义权重因子或提供额外处理逻辑的方式来改变文档得分。这类查询的一个很好的例子是 function_score 查询。function_score 查询可以让我们使用函数,从而通过数学计算的方式改变文档得分。举个例子,如果你希望离给定地理定位点越近的文档得分越高,则 function_score 查询可以帮助实现这个目的。

本类查询包括: boosting, constant_score, function_score, indices 查询。

8. 位置敏感查询

这类查询不仅可以匹配特定词项,还能匹配词项的位置信息。Elasticsearch 提供的各种范围查询就是这类查询的典型代表。我们还可以把 match_phrase 查询归入本类,因为至少从某种程度上来说,它也需要考虑被索引词项的位置信息。如果需要找出一组和其他单词保持一定距离的单词,比如“找出以下文档,同时包含 mastering 和 Elasticsearch 且这两个单词相互临近,其后不超过距离 3 的位置包含 second 和 edition 单词”,可以使用各种范围查询。不过,需要注意的是,这些范围查询将在未来版本的 Lucene 库中将被移除,届时 Elasticsearch 也不再提供支持。这是因为这些查询开销很大,需要消耗大量 CPU 资源才能保证正确处理。

本类查询包括: match_phrase, span_first, span_multi, span_near, span_not, span_or, span_term 查询。

9. 结构敏感查询

最后一类查询是结构敏感查询(structure aware query)。这类查询包括:

- ☐ nested 查询
- ☐ has_child 查询
- ☐ has_parent 查询
- ☐ top_children 查询

一般来说,所有支持对文档结构进行检索并且不需要对文档数据进行扁平化处理的查

询方式都可以归入此类。如果你正在寻找一种查询方式，能够在子文档或嵌套文档中进行查询，或查找属于给定父文档的子文档，则需要使用刚刚提及的查询方式之一。换句话说，如果需要处理文档中的数据关系，请选择使用这类查询。不过需要注意的是，尽管Elasticsearch可以支持一些数据关系，但它毕竟不是真正的关系数据库。

2.5.2 使用示例

既然我们已经了解了各类查询方式的适用场合以及期望结果，现在可以趁热打铁，用具体的使用示例来进一步加强对它们的认知。注意，这些例子并不能覆盖Elasticsearch查询的方方面面，而仅仅是对于我们通过查询获取所需信息的一些简单示例说明。

1. 测试数据

为了达到本节的目的，我们给library索引加入了两个新文档。

首先，我们需要微调一下索引映射以支持嵌套文档（本节某些例子中需要用到）。修改映射的命令如下：

```
curl -XPUT 'http://localhost:9200/library/_mapping/book' -d '{
  "book" : {
    "properties" : {
      "review" : {
        "type" : "nested",
        "properties": {
          "nickname" : { "type" : "string" },
          "text" : { "type" : "string" },
          "stars" : { "type" : "integer" }
        }
      }
    }
  }
}
```

然后，接着索引两个新文档。相关命令如下：

```
curl -XPOST 'localhost:9200/library/book/5' -d '{
  "title" : "The Sorrows of Young Werther",
  "author" : "Johann Wolfgang von Goethe",
  "available" : true,
  "characters" : ["Werther",
    "Lotte", "Albert",
    " Fräulein von B"],
  "copies" : 1,
```

```

    "otitle" : "Die Leiden des jungen Werthers",
    "section" : 4,
    "tags" : ["novel", "classics"],
    "year" : 1774,
    "review" : [{ "nickname" : "Anna", "text" : "Could be good, but not
my style", "stars" : 3}]
  },
  curl -XPOST 'localhost:9200/library/book/6' -d '{
    "title" : "The Peasants",
    "author" : "Władysław Reymont",
    "available" : true,
    "characters" : ["Maciej Boryna", "Jankiel", "Jagna Paczesiówna",
    "Antek Boryna"],
    "copies" : 4,
    "otitle" : "Chłopi",
    "section" : 4,
    "tags" : ["novel", "polish", "classics"],
    "year" : 1904,
    "review" : [{ "nickname" : "anonymous", "text" : "awesome
book", "stars" : 5}, { "nickname" : "Jane", "text" : "Great book, but
too long", "stars" : 4}, { "nickname" : "Rick", "text" : "Why bother,
when you can find it on the internet", "stars" : 3}]
  },

```

2. 基本查询示例

让我们看一下使用基本查询类别的例子。

(1) 查询给定范围的数据

匹配给定取值范围的文档的查询是最简单的查询方式之一。通常，这种查询作为一个更复杂查询或过滤器的一部分而存在。举例来说，一个可以查出副本数在 [1,3] 区间的书籍的查询如下所示：

```

curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "range" : {
      "copies" : {
        "gte" : 1,
        "lte" : 3
      }
    }
  }
}
```

(2) 简化的多词项查询

想象一个场景，用户需要传入一组书籍标签，期望查询出匹配这些标签的书籍。还有一个条件，如果用户给出的标签超过 3 个，则只要求至少 75% 的给定标签与索引中的书籍匹配。通常我们可以通过 bool 查询去实现这个目的，不过 Elasticsearch 提供了可以实现相同目的的 terms 查询。执行查询的命令如下：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query": {
    "terms": {
      "tags": [ "novel", "polish", "classics", "criminal", "new" ],
      "minimum_should_match": "3<75%"
    }
  }
}'
```

3. 组合查询示例

现在我们看看如何使用组合查询来组合其他查询方式。

(1) 对匹配文档加权

最简单的示例是使用包含一个可选的加权片段的 bool 查询来实现对部分文档的权重提升。举例来说，如果需要找出所有至少拥有一个副本的书籍，并对 1950 年后出版的书籍进行加权，可以使用如下查询命令：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query": {
    "bool": {
      "must": [
        {
          "range": {
            "copies": {
              "gte": 1
            }
          }
        }
      ],
      "should": [
        {
          "range": {
            "year": {
              "gt": 1950
            }
          }
        }
      ]
    }
  }
}'
```

```

    }
  }
}
}
}'

```

(2) 忽略查询的较低得分部分

我们之前提到的 `dis_max` 查询可以控制查询中较低得分部分的影响。举例来说，如果我们期望找出所有 `title` 字段匹配 “crime punishment” 或 `characters` 字段匹配 “raskolnikov” 的文档，并在文档打分时仅考虑得分最高的查询片段，可以执行如下查询命令：

```

curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "fields" : [ "_id", "_score" ],
  "query" : {
    "dis_max" : {
      "tie_breaker" : 0.0,
      "queries" : [
        {
          "match" : {
            "title" : "crime punishment"
          }
        },
        {
          "match" : {
            "characters" : "raskolnikov"
          }
        }
      ]
    }
  }
}'

```

查询结果如下：

```

{
  "took" : 3,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {

```



```

    "total" : 1,
    "max_score" : 0.2169777,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "4",
      "_score" : 0.2169777,
      "fields" : {
        "_id" : "4"
      }
    } ]
  }
}

```

我们来单独看一下查询各部分的打分。可以单独执行如下查询片段：

```

curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "fields" : [ "_id", "_score" ],
  "query" : {
    "match" : {
      "title" : "crime punishment"
    }
  }
}'

```

查询结果如下：

```

{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.2169777,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "4",
      "_score" : 0.2169777,
      "fields" : {
        "_id" : "4"
      }
    } ]
  }
}

```

单独执行下一个查询片段的命令如下：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "fields" : [ "_id", "_score" ],
  "query" : {
    "match" : {
      "characters" : "raskolnikov"
    }
  }
}'
```

查询结果如下：

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.15342641,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "4",
      "_score" : 0.15342641,
      "fields" : {
        "_id" : "4"
      }
    } ]
  }
}
```

可以看出，dis_max 查询返回的文档得分等于打分最高的查询片段的得分（上面的第一个查询片段）。这是因为我们设置 tie_breaker 属性为 0.0。

4. 无分析查询示例

让我们看看两个不使用任何分析器的查询示例。

（1）找出符合标签的结果

Elasticsearch 提供的 term 查询是最简单的无分析查询之一。我们一般很少单独使用 term 查询，而是常常将其使用在各种复合查询中。举个例子，假设我们想要查找出所有 tags 字段包含 “novel” 值的书籍。为了达到这个目的，需要执行如下查询命令：



```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "term" : {
      "tags" : "novel"
    }
  }
}'
```

(2) 在查询时高效处理停用词

Elasticsearch 提供了普通的 terms 查询，可以在查询时用一种高效的方式处理停用词。它将查询词项分成两组——重要的词项和不重要词项。重要词项的出现频率较低，相反，不重要词项的出现频率很高。Elasticsearch 首先用重要词项执行查询并计算文档得分，然后再使用不重要词项执行查询，这时不再计算文档得分。因此查询可以变得更快。

举例来说，以下两个查询单就查询结果而言非常相似，而结果的打分却不一样。注意，如果想清楚地看出两者打分的不同，我们需要准备大量的数据样本，并在索引时禁用停用词。

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "common" : {
      "title" : {
        "query" : "the western front",
        "cutoff_frequency" : 0.1,
        "low_freq_operator": "and"
      }
    }
  }
}'
```

第二个查询如下：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "bool" : {
      "must" : [
        {
          "term" : { "title" : "western" }
        },
        {
          "term" : { "title" : "front" }
        }
      ],
      "should" : [
        {
          "term" : { "title" : "the" }
        }
      ]
    }
  }
}'
```



5. 全文检索查询示例

全文检索是一个宽泛的主题，其使用场景也十分广泛。在这里我们选出两个简单场景的查询示例加以展示。

(1) 使用 Lucene 查询语法

某些时候，使用 Lucene 查询语法是不错的选择。我们曾在 1.1.4 节介绍过 Lucene 查询语法。举个例子，假如我们想找出 title 字段包含 “sorrows” 和 “young” 词项、author 字段包含 “von goethe” 短语，并且副本数不超过 5 个的文档，可以执行如下查询：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "query_string" : {
      "query" : "+title:sorrows +title:young +author:\"von goethe\" -
copies:[5 TO *]"
    }
  }
}'
```

在这个查询中，我们使用了 Lucene 查询语法来传递所有匹配条件，让 Lucene 通过查询解析器来构造合适的查询。

(2) 对用户查询串进行容错处理

在某些场景下，来自用户的查询可能包含错误。比如，下面这个查询：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "query_string" : {
      "query" : "+sorrows +young \"",
      "default_field" : "title"
    }
  }
}'
```

Elasticsearch 将返回如下响应：

```
"error" : "SearchPhaseExecutionException[Failed to execute phase
[query]]"
```

这意味着在构建查询时遇到了解析错误，查询无法被成功地构建出来。这也是 Elasticsearch 引入 simple_query_string 查询的原因。它使用一个可尝试处理用户输入错误的查询解析器，并试图猜测用户的查询用意。如果用 simple_query_string 查询来改写上面这个例子，代码如下：




```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "simple_query_string" : {
      "query" : "+sorrows +young \"",
      "fields" : [ "title" ]
    }
  }
}'
```

如果执行这个查询，你将看到 Elasticsearch 能够返回合适的文档结果，尽管查询并未被恰当构造。

6. 模式匹配查询示例

模式匹配查询的例子很多，不过在这里我们只打算展示其中两个。

(1) 使用前缀查询实现自动完成功能

针对索引数据提供自动完成功能是一种常见的应用场景。如我们所知，前缀查询不会被分析，直接工作于特定字段中被索引的词项上。因此，实际的功能依赖于索引时生成词条的方式。举例来说，假定我们希望针对 title 字段的所有词条提供自动完成功能。此时用户输入的前缀是“wes”，符合条件的对应查询构造如下：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "prefix" : {
      "title" : "wes"
    }
  }
}'
```

(2) 模式匹配

如果我们想匹配特定模式，而此时索引中的词条无法支持，可以尝试使用 regexp 查询。读者需要注意的是，这种查询非常昂贵，请尽量避免使用。当然，有时候我们不得不使用它。还有一点需要注意的是，regexp 查询的执行性能与所选正则表达式相关。如果你选择了一个能够被改写成大量词项的正则表达式，执行性能将极其糟糕。

现在来看一下使用 regexp 查询的例子。假定我们需要找出符合以下条件的文档：文档的 characters 字段中包含以“wat”开头、以“n”结尾、中间有两个任意字符的词项。为了实现这些条件，可以使用类似下面的 regexp 查询命令：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "regexp" : {
```



```

    "characters" : "wat..n"
  }
}
}'

```

7. 支持相似度操作的查询示例

让我们看两个关于如何查找近似文档和词项的简单示例。

(1) 找出给定词项的近似词项

一个非常简单的例子是使用 fuzzy 查询找出包含给定词项近似词项的文档。比如，如果我们需査找包含“crimea”的近似词项的文档，可以执行如下查询命令：

```

curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "fuzzy" : {
      "title" : {
        "value" : "crimea",
        "fuzziness" : 3,
        "max_expansions" : 50
      }
    }
  }
}'

```

(2) 找出拥有近似字段值的文档

另一个相似度査询的案例是，根据我们在査询中提供的字段值，找出包含类似字段值的文档。比如，我们想找出 title 字段值类似“western front battles”的书籍，可以构造如下査询：

```

curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "fuzzy_like_this_field" : {
      "title" : {
        "like_text" : "western front battles",
        "max_query_terms" : 5
      }
    }
  }
}'

```

査询结果如下：

```

{
  "took" : 10,
  "timed_out" : false,

```



```

    "_shards" : {
      "total" : 5,
      "successful" : 5,
      "failed" : 0
    },
    "hits" : {
      "total" : 2,
      "max_score" : 1.0162667,
      "hits" : [ {
        "_index" : "library",
        "_type" : "book",
        "_id" : "1",
        "_score" : 1.0162667,
        "_source":{ "title": "All Quiet on the Western
Front","otitle": "Im Westen nichts Neues","author": "Erich
Maria Remarque","year": 1929,"characters": ["Paul B|umer",
"Albert Kropp", "Haie Westhus", "Fredrich M|dler",
"Stanislaus Katczinsky", "Tjaden"],"tags":
["novel"],"copies": 1,
"available": true, "section" : 3}
      }, {
        "_index" : "library",
        "_type" : "book",
        "_id" : "5",
        "_score" : 0.4375,
        "_source":{ "title" : "The Sorrows of Young Werther","author"
: "Johann Wolfgang von Goethe","available" :
true,"characters" : ["Werther","Lotte","Albert","Fraulein
von B"],"copies" : 1, "otitle" : "Die Leiden des jungen
Werthers","section" : 4,"tags" : ["novel",
"classics"],"year" : 1774,"review" : [{"nickname" :
"Anna","text" : "Could be good, but not my style","stars" :
3}]}
      } ]
    }
  }
}

```

从上面结果可见，有时候查询结果跟我们的期望有些出入（比如结果中第2本书的标题）。这是因为 Elasticsearch 认为它们之间有相似性。在前面这个查询中，Elasticsearch 将所有词项执行一次模糊查询，然后为匹配的文档选择出一组最佳查分词项。

8. 支持打分操作的查询示例

涉及相关度，Elasticsearch 提供了一些可以按需修改文档得分的查询。当然，大多数查询方式都支持权重，可以让我们拥有更多的操作余地。接下来让我们看看两个支持打分操作的查询示例。

(1) 偏爱新书

假定我们更喜欢新出版的书籍，因此 1986 年出版的书籍要比 1870 年出版的书籍拥有更高的得分。满足这个需求的查询命令如下：



```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "function_score" : {
      "query" : {
        "match_all" : {}
      },
      "score_mode" : "multiply",
      "functions" : [
        {
          "gauss" : {
            "year" : {
              "origin" : 2014,
              "scale" : 2014,
              "offset" : 0,
              "decay": 0.5
            }
          }
        }
      ]
    }
  }
}
```

我们将在第3章介绍 `function_score` 查询。在这里，如果你仔细观察刚才这个查询的响应结果，可以发现越新的书籍得分越高。

（2）对拥有某些值的书籍扣分

有时候，我们需要降低某些文档的重要性，却依然要在结果列表中输出它们。举个例子，我们可能想要列出所有的书籍，不过要通过降低书籍得分的方式把那些当前无货的书籍放到结果列表的末尾。我们不希望按标记是否有货的字段进行排序，因为用户有时候清楚地知道他要找什么书，因此针对全文检索查询的结果得分是很重要的。不过，如果仅仅想把当前无货的书籍排到结果尾部，可以执行如下查询命令：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "boosting" : {
      "positive" : {
        "match_all" : {}
      },
      "negative" : {
        "term" : {
          "available" : false
        }
      }
    }
  }
}
```




```

    },
    "negative_boost" : 0.2
  }
}
}'

```

9. 位置敏感查询示例

位置敏感查询允许我们匹配包含正确次序短语和词项的文档。这类查询因为资源占用问题，不经常被使用。让我们看两个例子。

(1) 匹配短语

匹配短语的 `match_phrase` 查询是最简单的位置敏感查询，也是本类别中使用最多的。举例来说，`otitle` 字段匹配短语“leiden des jungen”的查询命令如下：

```

curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "match_phrase" : {
      "otitle" : "leiden des jungen"
    }
  }
}'

```

(2) 处处可见范围查询

当然，短语查询在处理位置敏感需求时非常简便。不过，如果我们想执行一个查询，找出符合以下条件的文档：在“die”词项后面不超过两个位置的地方包含一个“des jungen”短语，并且紧跟着短语后面是一个“werthers”词项。这时候，该怎么办呢？我们可以使用范围查询。符合这些条件的查询命令类似如下：

```

curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "span_near" : {
      "clauses" : [
        {
          "span_near" : {
            "clauses" : [
              {
                "span_term" : {
                  "otitle" : "die"
                }
              },
              {
                "span_near" : {
                  "clauses" : [

```



(1) 返回包含某个嵌套子文档的父文档

第一个例子非常简单。假定我们要查找所有拥有 4 星及以上评论的书籍。相应的查询命令如下：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "nested" : {
      "path" : "review",
      "query" : {
        "range" : {
          "stars" : {
            "gte" : 4
          }
        }
      }
    }
  }
}
```

(2) 嵌套子文档的得分影响父文档得分

假定我们要返回所有拥有评论的书籍，并且按照最高评论星级对这些书籍进行排序。满足这些条件的查询命令如下：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "nested" : {
      "path" : "review",
      "score_mode" : "max",
      "query" : {
        "function_score" : {
          "query" : { "match_all" : {} },
          "score_mode" : "max",
          "boost_mode" : "replace",
          "field_value_factor" : {
            "field" : "stars",
            "factor" : 1,
            "modifier" : "none"
          }
        }
      }
    }
  }
}
```

2.6 小结

在本章中，我们了解了 Apache Lucene 默认的打分机制是如何运作的，探讨了查询改写的处理过程——查询改写是如何实现的以及为什么需要查询改写。我们还认识了查询模板的工作原理，以及它们是如何简化查询构建的。我们还一起探索了不同的查询过滤方式、它们之间的差异，以及它们的使用时机。最后，我们把查询指派到不同的分组中，并学习了不同分组的使用场景和具体示例。

下一章中，我们将告别全文检索，把目光投向其他查询功能上。首先我们将把知识面拓展到二次评分功能，并具备对搜索结果前 N 个文档重新打分的能力。然后我们将学习如何加载重要词项，并使用聚集功能实现文档分组。我们还将对比父子关系和嵌套文档之间的差异，掌握方法查询的使用。最后，我们还将学习如何高效地对结果文档集进行分页。

不只是文本搜索

在上一章中，我们着重讨论了 Elasticsearch 的各种查询。本章将开始介绍默认 Apache Lucene 评分（scoring）、过滤（filtering）的工作原理，同时也将讨论在特定场合应选用何种查询。在本章中，我们也会继续讨论 Elasticsearch 的一些与查询和数据分析都相关的功能。到本章结束时，将涵盖以下内容：

- ❑ 什么是查询二次评分，如何利用它优化查询以及为某些文档重新打分
- ❑ 控制 multimatching
- ❑ 分析数据，从中获取有意义的词项（集）
- ❑ 使用 Elasticsearch 对文档分组
- ❑ 使用对象、嵌套文档、parent-child 功能等处理关系类型数据时的差异
- ❑ Elasticsearch 脚本的使用，如 Groovy 及 Lucene 表达式等

3.1 查询二次评分

Elasticsearch 提供的关键特性中就包括了查询二次评分（query rescoring），它能改变某个查询执行后返回文档的得分，自然而然地也能改变这些文档的排序。Elasticsearch 只使用了一个简单的技巧，它只对返回文档中的 top-*N* 进行二次评分，即只改变部分返回文档的排序结果。这么做的理由有很多。其中一种就是基于性能方面的考虑，如评分时使用了脚本，而脚本非常耗时，作为折中，可以仅对返回文档的一个子集进行重新打分。读者能想象得到，二次评分使用户有很多机会定制业务逻辑。现在，让我们深入了解一下这项功能，并

分析能从中获得什么便利。

3.1.1 什么是查询二次评分

查询二次评分是 Elasticsearch 中的一种机制，能对查询的返回文档的前若干个文档重新打分。这意味着 Elasticsearch 先取得某个查询（或 `post_filter` 短语）的命中文档的前 N 个，然后执行某个公式为这些文档重新打分。以 `Term Query` 为例，首先执行查询，得到该查询的命中文档，然后对命中文档的前 100 个进行重新打分，而不是对所有的命中文档重新打分。请记住，如果 `search_type` 为 `scan` 或 `count`，二次评分操作将不会被执行。这意味着在类似案例中，二次评分功能不予考虑。

3.1.2 一个查询例子

下面是一个简单的查询例子：

```
{
  "fields" : ["title", "available"],
  "query" : {
    "match_all" : {}
  }
}
```

该查询执行后将命中所有文档。因为使用了 `match_all` 查询，所有命中文档的得分都等于 1.0。这个查询非常简单，但足以用来演示查询二次评分对检索结果的影响。

3.1.3 二次评分查询的结构

现在，我们将使用查询二次评分的功能来改写前面的查询。改写逻辑很简单，就是将文档得分改为文档的 `year` 字段的值。修改后的查询如下所示：

```
{
  "fields": ["title", "available"],
  "query": {
    "match_all": {}
  },
  "rescore": {
    "query": {
      "rescore_query": {
        "function_score": {
          "query": {
            "match_all": {}
          },
          "script_score": {
            "script": "doc['year'].value"
          }
        }
      }
    }
  }
}
```

```

    }
  }
}
}

```



注意

读者请记住，如果使用 Elasticsearch 1.4 或更老的版本，使用者需要将前面的查询的 `lang` 属性设置为 `groovy`。进一步说明一下，上面的例子在 Elasticsearch 1.3.8 及 1.4.3 版本之前都可以使用 `groovy` 动态脚本，在版本 1.2 之前可使用 `MVEL` 脚本。如果想通过 `groovy` 使用动态脚本，可设置 `Elasticsearch.yml` 文件的 `script.groovy.sandbox.enabled` 的属性为 `true`。但是需要记住，开启该功能会有安全风险。

进一步考察前面的查询。首先要注意的是 `rescore` 对象，该对象持有一个查询，它将对特定查询的命中文档的得分产生影响。在我们的案例中，此影响方式非常简单，就是将文档得分改写为文档的 `year` 字段的值。读者还要注意，如果使用 `curl` 客户端，需要对脚本值进行转义，如 `doc['year'].value` 应转义为 `doc["year"].value`。



注意

前面的例子中，在 `rescore` 对象的内部，可以看到有一个 `query` 对象。在本书撰写时，只有查询对象可用。随着 Elasticsearch 版本的演进，会有其他更多的选项可影响文档得分的计算。

如果将上面的查询保存在 `query.json` 文件中，可使用下面的命令来执行该查询：

```
curl localhost:9200/library/book/_search?pretty -d @query.json
```

返回结果与下面类似（请注意这里略去了响应消息的结构信息）：

```

{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 6,
    "max_score" : 1962.0,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "2",
      "_score" : 1962.0,
      "fields" : {
        "title" : [ "Catch-22" ],
        "available" : [ false ]
      }
    }
  ]
}

```

```

    }, {
      "_index" : "library",
      "_type" : "book",
      "_id" : "3",
      "_score" : 1937.0,
      "fields" : {
        "title" : [ "The Complete Sherlock Holmes" ],
        "available" : [ false ]
      }
    }, {
      "_index" : "library",
      "_type" : "book",
      "_id" : "1",
      "_score" : 1930.0,
      "fields" : {
        "title" : [ "All Quiet on the Western Front" ],
        "available" : [ true ]
      }
    }, {
      "_index" : "library",
      "_type" : "book",
      "_id" : "6",
      "_score" : 1905.0,
      "fields" : {
        "title" : [ "The Peasants" ],
        "available" : [ true ]
      }
    }, {
      "_index" : "library",
      "_type" : "book",
      "_id" : "4",
      "_score" : 1887.0,
      "fields" : {
        "title" : [ "Crime and Punishment" ],
        "available" : [ true ]
      }
    }, {
      "_index" : "library",
      "_type" : "book",
      "_id" : "5",
      "_score" : 1775.0,
      "fields" : {
        "title" : [ "The Sorrows of Young Werther" ],
        "available" : [ true ]
      }
    }
  ]
}

```

如你所见, Elasticsearch 执行了第一个查询, 并召回了所有文档。进一步查看文档得分, 此时发现 Elasticsearch 已使用第 2 个查询对第 1 个查询的前 N 个命中文档进行重新打分了。最终, 这些被重新打分的文档的得分为两个查询下的得分之和。

读者不难理解, 如果性能需要重点注意, 那么对脚本的使用需要谨慎。这就是为什么

要在 `rescore` 阶段使用脚本。如果第 1 个查询命中成千上万的文档，在此阶段对所有文档使用脚本将会导致极其糟糕的性能。因为 `rescore` 只对返回文档的 `top-N` 的文档进行打分，因此极大地缓解了性能问题。



注意 在我们的例子中，只有一个 `rescore` 的定义，但是从 Elasticsearch 1.1 版本开始，有多个 `rescore` 查询可用于修改返回结果的得分。因此可以构建多层次的查询来改变返回结果的 `top-N` 文档的得分与排序，每个 `rescore` 查询的处理结果可以作为下一个 `rescore` 查询的输入。

现在再来看看如何对查询二次评分调优，以及有哪些可用的参数。

3.1.4 二次评分参数

可对 `rescore` 对象中的查询使用下面这些参数。

- ❑ `window_size` (默认为 `from`、`size` 参数之和)：该参数指定了每个 `shard` 中需要二次评分的文档个数。
- ❑ `query_weight` (默认为 1)：第 1 个查询的得分将乘以该参数值，之后再与二次评分查询得分相加。
- ❑ `rescore_query_weight` (默认为 1)：在与第 1 个查询得分相加之前，二次评分查询得分将乘以该参数值。

换句话说，文档最终得分公式如下：

$$\text{original_query_score} * \text{query_weight} + \text{rescore_query_score} * \text{rescore_query_weight}$$

选择评分模式

默认情况下，被改写的文档的得分为两个查询下的得分之和。可设置 `score_mode` 来修改评分模式。该参数有下面这些选项。

- ❑ `total`：文档得分为两个查询下的得分之和。
- ❑ `multiply`：文档得分为两个查询下的得分之积。
- ❑ `avg`：文档得分为两个查询下得分的平均值。
- ❑ `max`：文档得分为两个查询下得分之中的最大值。
- ❑ `min`：文档得分为两个查询下得分之中的最小值。

3.1.5 总结

有的时候，也许我们想干预返回结果中第 1 页文档的排序，使之按某种规则排序。不

幸的是，并不能通过二次评分功能来实现这个目的。读者可能第一时间想到了 `window_size` 参数，而事实上该参数与返回结果的第 1 页并无关联，它用于指定每个 shard 返回文档的个数。此外 `window_size` 不能小于 `page size`（如果 `windows_size` 的值小于 `page size`，则被设置为 `page size` 的值）。另外有件事情非常重要，二次评分并不能与排序（`sorting`）结合使用，这是因为排序在重新打分之前就结束了，因此排序并不能预知文档得分的变化。

3.2 多匹配控制

在 Elasticsearch 1.1 版本中，只能对 `multi_match` 查询做有限的控制。我们当然能设置 `query` 要查询的字段，同样也能设置析取（`disjunction`）查询的最大数量（通过设置 `use_dis_max` 参数）。最后，可以通过设置权重告诉 Elasticsearch 每个字段的重要程度。下面是一个在多个字段上执行查询的范例：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "multi_match" : {
      "query" : "complete conan doyle",
      "fields" : [ "title^20", "author^10", "characters" ]
    }
  }
}'
```

针对上面这个简单的查询，命中任意指定字段的任意文档都将被召回。除此之外，上面查询中 `title` 字段权重最高，其次是 `author` 字段。

当然，我们也可以使用最大析取查询（`disjunction max query`）：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "multi_match" : {
      "query" : "complete conan doyle",
      "fields" : [ "title^20", "author^10", "characters" ],
      "use_dis_max" : true
    }
  }
}'
```

最大析取查询除了对文档得分计算有影响外，其他方面作用不大。

多匹配类型

自 Elasticsearch 1.1 版本发布以后，`use_dis_max` 属性被弃用，同时 Elasticsearch 又提

供了一种新属性：`type`。可使用该属性决定 `multi_match` 查询内部的执行方式。现在我们来探究一下有哪些可能的方式来控制 Elasticsearch 的多字段查询。



注意 请记住，`tie_breaker` 并未被废弃，不用担心未来使用该属性时的兼容性问题。

1. `best_fields` 匹配

使用这种查询匹配类型（`best field matching`），需要将 `multi_match` 查询的 `type` 属性值设置为 `best_fields` 查询。此时 `multi_match` 查询会为 `fields` 字段中的每个字段生成一个查询。这种查询匹配类型特别适合有多字段且 `query` 文本相同的最佳匹配查询。可查看下面这个查询范例：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "multi_match" : {
      "query" : "complete conan doyle",
      "fields" : [ "title", "author", "characters" ],
      "type" : "best_fields",
      "tie_breaker" : 0.8
    }
  }
}'
```

前面的查询有更规整的表述方式：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "dis_max" : {
      "queries" : [
        {
          "match" : {
            "title" : "complete conan doyle"
          }
        },
        {
          "match" : {
            "author" : "complete conan doyle"
          }
        },
        {
          "match" : {
            "characters" : "complete conan doyle"
          }
        }
      ]
    }
  }
}'
```

```

    }
  }
],
"tie_breaker" : 0.8
}
}
}'

```

观察这两个查询的返回结果，你可能会注意到下面的内容：

```

{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.033352755,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "3",
      "_score" : 0.033352755,
      "_source":{ "title": "The Complete Sherlock
Holmes","author": "Arthur Conan Doyle","year":
1936,"characters": ["Sherlock Holmes","Dr. Watson", "G.
Lestrade"],"tags": [],"copies": 0, "available" : false,
"section" : 12}
    } ]
  }
}

```

读者不难发现，这两个查询的返回文档完全一样，文档得分也一样。读者需要注意的是文档得分的计算方式。如果查询中 `tie_breaker` 属性被设置，则文档得分等于最佳匹配字段得分与其他匹配字段的得分之和，只是其他被匹配上的字段得分需要乘以 `tie_breaker` 的值。如果 `tie_breaker` 属性没有被设置，则文档得分等于最佳匹配字段的得分。

此外，值得一提的是最佳字段匹配的原理：当我们使用 AND 操作符或 `minimum_should_match` 属性时，会发生什么事情？答案很简单，最佳字段匹配被转换为多个 `match` 查询，并且 `operator`、`minimum_should_match` 的属性值被应用到这些被生成的 `match` 查询上。由于这个原因，下面的查询将不会返回任何命中文档：

```

curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {

```



```

"multi_match" : {
  "query" : "complete conan doyle",
  "fields" : [ "title", "author", "characters" ],
  "type" : "best_fields",
  "operator" : "and"
}
}
}'

```

这是由于上面的查询被转换为下面这样了：

```

curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "dis_max" : {
      "queries" : [
        {
          "match" : {
            "title" : {
              "query" : "complete conan doyle",
              "operator" : "and"
            }
          }
        },
        {
          "match" : {
            "author" : {
              "query" : "complete conan doyle",
              "operator" : "and"
            }
          }
        },
        {
          "match" : {
            "characters" : {
              "query" : "complete conan doyle",
              "operator" : "and"
            }
          }
        }
      ]
    }
  }
}'

```

而这个查询与 Lucence 中的这个复合查询是等价的：

```
(+title:complete +title:conan +title:doyle) | (+author:complete
+author:conan +author:doyle) | (+characters:complete
+characters:conan +characters:doyle)
```

事实上，索引中并没有任意文档在单个字段中包含了 complete、conan、doyle 这 3 个词项。还好天无绝人之路，我们可以使用 cross-fields（跨字段）匹配来实现在多个字段中命中（不同的）词项。

2. cross_fields 匹配

如果期望查询中所有词项都在命中文档中出现，那么使用 cross_fields 匹配是非常合适的。来回忆一下上一个查询范例，这里只是用 cross_fields 匹配类型替换其中了 best_fields 匹配类型，如下所示：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "multi_match" : {
      "query" : "complete conan doyle",
      "fields" : [ "title", "author", "characters" ],
      "type" : "cross_fields",
      "operator" : "and"
    }
  }
}'
```

此时 Elasticsearch 返回结果是这样的：

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.08154379,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "3",
      "_score" : 0.08154379,
      "_source":{ "title": "The Complete Sherlock
Holmes","author": "Arthur Conan Doyle","year":
1936,"characters": ["Sherlock Holmes","Dr. Watson", "G.
Lestrade"],"tags": [],"copies": 0, "available" : false,
"section" : 12}
    } ]
  }
}
```

这是因为查询被转化为如下等价的 Lucene 查询：

```
+(title:complete author:complete characters:complete)
+(title:conan author:conan characters:conan) +(title:doyle
author:doyle characters:doyle)
```

此时只有命中所有词项（任意字段）的文档才被返回。当然，这是使用 AND 操作符时的搜索结果，如果使用 OR 操作符，在任意字段中命中了至少一个词项的文档会被召回。

当使用 `cross_fields` 类型时，有件事情需要特别引起注意，就是不同字段的词项频率可能会带来问题。Elasticsearch 对查询中涉及的多个字段中的词项频率做了平衡。简单来说，在查询涉及字段中，为每个命中词项赋予近似的权重。

3. `most_fields` 匹配

另一种可用的 `multi_field` 选项为 `most_fields` 类型。根据官方文档所述，该匹配类型用于帮助检索那些多处包含相同文本，但是文本分析处理方式不同的文档。典型例子就是多语言处理（多字段）。例如，我们想搜索 `title` 或 `original title` 字段中包含 `die leiden` 的文档，可执行下面这个查询：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "multi_match" : {
      "query" : "Die Leiden",
      "fields" : [ "title", "otitle" ],
      "type" : "most_fields"
    }
  }
}'
```

在 Elasticsearch 内部，上面的查询被转换为如下查询：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "bool" : {
      "should" : [
        {
          "match" : {
            "title" : "die leiden"
          }
        },
        {
          "match" : {
            "otitle" : "die leiden"
          }
        }
      ]
    }
  }
}'
```

```

    }
  ]
}
}'

```

4. phrase 匹配

接下来要介绍的是 phrase 匹配类型，它与前面提到的 best_fields 匹配类型非常类似。区别在于，后者将原始查询转换为 match 查询，而前者将原始查询转换为 match_phrase 查询。可以查看下面的查询范例：

```

curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "multi_match" : {
      "query" : "sherlock holmes",
      "fields" : [ "title", "author" ],
      "type" : "phrase"
    }
  }
}'

```

因为使用了 phrase 匹配，上面的查询被转换为下面这种形式：

```

curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "dis_max" : {
      "queries" : [
        {
          "match_phrase" : {
            "title" : "sherlock holmes"
          }
        },
        {
          "match_phrase" : {
            "author" : "sherlock holmes"
          }
        }
      ]
    }
  }
}'

```


5. phrase with prefixes 匹配

该类型与 phrase 类型原理完全一致，只是原始查询被转换为 match_phrase_prefix 查询，而不是 match_phrase 查询。读者可试运行下面这个查询：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "multi_match" : {
      "query" : "sherlock hol",
      "fields" : [ "title", "author" ],
      "type" : "phrase_prefix"
    }
  }
}'
```

在 Elasticsearch 内部，原始查询被转换为类似下面这样的查询：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query" : {
    "dis_max" : {
      "queries" : [
        {
          "match_phrase_prefix" : {
            "title" : "sherlock hol"
          }
        },
        {
          "match_phrase_prefix" : {
            "author" : "sherlock hol"
          }
        }
      ]
    }
  }
}'
```

截至现在，我们已经探讨了 multi_match 查询的 type 属性的各种可能的选项。使用者无须构造复杂的查询，就能获取各种所需的结果。Elasticsearch 会自行搞定评分及相关问题。

3.3 重要词项聚合

Elasticsearch 1.0 之后新增了多种聚合类型，其中之一就是 significant_terms 聚合，从 Elasticsearch 1.1 版本之后它就可以使用了。该聚合能帮助用户获取跟指定查询高度相关的

词项(集)。这种聚合的意义在于不仅能从查询返回结果中提取与查询最相关的一组词项,并且能找出最相关的那个词项。

这种聚合用途广泛,既可以帮助用户找到应用环境中最不堪重负的服務器,也可以帮助用户从文本中提取人名绰号推荐项。更引人入胜的是,Elasticsearch 可以监控词典中词项重要度的变化,重要度提升比较明显的就被推荐为候选重要词项。



注意 significant_terms 聚合被官方标记为实验性质的,可能在未来的版本中被修改或移除。

3.3.1 一个例子

描述 significant_terms 聚合的最好方式当然是通过范例。现在,我们来索引 12 个简单的文档,这些文档代表了对实习生的工作评价(索引命令保存在 significant.sh 脚本中,便于在 Linux 平台上执行):

```
curl -XPOST 'localhost:9200/interns/review/1' -d '{"intern" :
  "Richard", "grade" : "bad", "type" : "grade"}'
curl -XPOST 'localhost:9200/interns/review/2' -d '{"intern" : "Ralf",
  "grade" : "perfect", "type" : "grade"}'
curl -XPOST 'localhost:9200/interns/review/3' -d '{"intern" :
  "Richard", "grade" : "bad", "type" : "grade"}'
curl -XPOST 'localhost:9200/interns/review/4' -d '{"intern" :
  "Richard", "grade" : "bad", "type" : "review"}'
curl -XPOST 'localhost:9200/interns/review/5' -d '{"intern" :
  "Richard", "grade" : "good", "type" : "grade"}'
curl -XPOST 'localhost:9200/interns/review/6' -d '{"intern" : "Ralf",
  "grade" : "good", "type" : "grade"}'
curl -XPOST 'localhost:9200/interns/review/7' -d '{"intern" : "Ralf",
  "grade" : "perfect", "type" : "review"}'
curl -XPOST 'localhost:9200/interns/review/8' -d '{"intern" :
  "Richard", "grade" : "medium", "type" : "review"}'
curl -XPOST 'localhost:9200/interns/review/9' -d '{"intern" :
  "Monica", "grade" : "medium", "type" : "grade"}'
curl -XPOST 'localhost:9200/interns/review/10' -d '{"intern" :
  "Monica", "grade" : "medium", "type" : "grade"}'
curl -XPOST 'localhost:9200/interns/review/11' -d '{"intern" :
  "Ralf", "grade" : "good", "type" : "grade"}'
curl -XPOST 'localhost:9200/interns/review/12' -d '{"intern" :
  "Ralf", "grade" : "good", "type" : "grade"}'
```

当然,为了演示 significant_terms 聚合的实力,固然可以使用更大的数据集。不过本书的目标是向读者传授 significant_terms 聚合的用法及原理。因此 12 个文档已经足够。

现在,让我们来查找针对 Richard 的典型评价。为了实现该目的,可使用下面这个查询:

```
curl -XGET 'localhost:9200/interns/_search?pretty' -d '{
  "query" : {
    "match" : {
      "intern" : "Richard"
    }
  },
  "aggregations" : {
    "description" : {
      "significant_terms" : {
        "field" : "grade"
      }
    }
  }
}'
```

查询结果如下:

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 5,
    "max_score" : 1.4054651,
    "hits" : [ {
      "_index" : "interns",
      "_type" : "review",
      "_id" : "4",
      "_score" : 1.4054651,
      "_source":{"intern" : "Richard", "grade" : "bad"}
    }, {
      "_index" : "interns",
      "_type" : "review",
      "_id" : "3",
      "_score" : 1.0,
      "_source":{"intern" : "Richard", "grade" : "bad"}
    }, {
      "_index" : "interns",
      "_type" : "review",
      "_id" : "8",
      "_score" : 1.0,
      "_source":{"intern" : "Richard", "grade" : "medium"}
    }, {
      "_index" : "interns",
      "_type" : "review",
      "_id" : "1",
```

```

    "_score" : 1.0,
    "_source" : { "intern" : "Richard", "grade" : "bad" }
  }, {
    "_index" : "interns",
    "_type" : "review",
    "_id" : "5",
    "_score" : 1.0,
    "_source" : { "intern" : "Richard", "grade" : "good" }
  } ]
},
"aggregations" : {
  "description" : {
    "doc_count" : 5,
    "buckets" : [ {
      "key" : "bad",
      "doc_count" : 3,
      "score" : 0.84,
      "bg_count" : 3
    } ]
  }
}
}
}

```

如读者所见，Elasticsearch 返回结果中，针对 Richard 的主流评价是 Bad。也许这个评价对他来说是不客观的，但是谁又知道呢。

3.3.2 选择重要词项

为了计算词项的重要度，Elasticsearch 会从两个数据集中查询词项重要度变化的报告信息。这两个数据集被称为前台数据集（foreground）和后台数据集（background）。前台数据集指的是查询返回的文档集，后台数据集指的是索引中的数据集（索引可能有多个，这依赖于我们查询数据源的设置）。假设某个词项仅在被索引的 100 万个文档中的 10 个文档中出现，但是在被召回的 10 个文档中的 5 个中出现，那么这个词项应该被定义为重要词项，需要重点对待。

现在我们再返回之前的范例，再深入分析一下。Richard 被打分者评价了 5 次，有 3 种得分，bad 三次，medium 一次，good 一次。这 3 种评分，bad 在查询的 5 个召回文档中的 3 个中出现了（参考 bg_count 属性），而索引中总共有 12 个文档（这里的索引为后台数据集）。因此 bad 出现的比例占召回文档的 60%。如读者所见，bad 在前台数据集中重要度相对后台数据集中重要度有大幅变化，因此 Elasticsearch 在 significant_terms 聚合结果中返回它。

3.3.3 多值分析

当然，significant_terms 聚合也可以嵌套使用，提供连接多个数据集的强大数据分析

能力。例如，我们想找出每个实习生的典型评分，可以在查询的 `terms` 聚合中嵌套使用 `significant_terms` 聚合。如下所示：

```
curl -XGET 'localhost:9200/interns/_search?size=0&pretty' -d '{
  "aggregations" : {
    "grades" : {
      "terms" : {
        "field" : "intern"
      },
      "aggregations" : {
        "significantGrades" : {
          "significant_terms" : {
            "field" : "grade"
          }
        }
      }
    }
  }
}
```

Elasticsearch 的返回结果类似下面这种：

```
{
  "took" : 71,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 12,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "grades" : {
      "doc_count_error_upper_bound" : 0,
      "sum_other_doc_count" : 0,
      "buckets" : [ {
        "key" : "ralf",
        "doc_count" : 5,
        "significantGrades" : {
          "doc_count" : 5,
          "buckets" : [ {
            "key" : "good",
            "doc_count" : 3,
            "score" : 0.21000000000000002,
```

```

        "bg_count" : 4
      } ]
    }
  }, {
    "key" : "richard",
    "doc_count" : 5,
    "significantGrades" : {
      "doc_count" : 5,
      "buckets" : [ {
        "key" : "bad",
        "doc_count" : 3,
        "score" : 0.6,
        "bg_count" : 3
      } ]
    }
  }, {
    "key" : "monica",
    "doc_count" : 2,
    "significantGrades" : {
      "doc_count" : 2,
      "buckets" : [ ]
    }
  } ]
}
}
}

```

如读者所见，我们得到了实习生 Ralf（key 属性值为 ralf）和 Richard（key 属性值为 richard）对应的结果，但没有 Monica 的任何信息。这是因为与 intern 字段中 monica 值关联的 grade 字段中的词项重要度没有显著改变。

重要词项聚合及全文搜索字段

当然，significant_terms 聚合可用于全文搜索字段，尤其是在用于确定文本关键字时。有件事情需要谨记，已分词字段的聚合非常耗费内存，因为需要把所有词项加载到内存中。

举个例子，我们在 library 索引的 title 字段中运行 significant_terms 聚合，如下所示：

```

curl -XGET 'localhost:9200/library/_search?size=0&pretty' -d '{
  "query" : {
    "term" : {
      "available" : true
    }
  },
  "aggregations" : {
    "description" : {
      "significant_terms" : {
        "field" : "title"
      }
    }
  }
}

```

```

    }
  }
},

```

然而，返回结果并没有给我们带来任何有用的信息：

```

{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 4,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "description" : {
      "doc_count" : 4,
      "buckets" : [ {
        "key" : "the",
        "doc_count" : 3,
        "score" : 1.125,
        "bg_count" : 3
      } ]
    }
  }
}

```

造成这种现象的原因是数据集不够大，造成返回结果效果不明显。仅从逻辑的角度来看，词项 the 是 title 字段中的重要词项。

3.3.4 额外的配置

对于 significant_terms 聚合的介绍，似乎是足够了。然而我们并不会就此画上句号，还会继续介绍该聚合类型的各种配置，使得用户能通过配置来控制内部计算方式，以满足应用需求。

1. 控制返回 bucket 数量

Elasticsearch 运行控制返回结果的最大 bucket 数量，可通过设置 size 属性来配置它。然而，最终返回的 bucket 链表的长度会大于 size 的值。如果词项个数大于 size 的值，就会出现这种情况。

如果读者想控制更多,可考虑设置 `shard_size` 属性。该属性确定每个 shard 返回多少个候选的重要词项。值得注意的是,低频词项通常是非常有意义的,而 Elasticsearch 在聚合节点上合并返回结果时没有考虑这个因素。因此,将 `shard_size` 属性值设置为大于 `size` 属性值是很有必要的。

还有一点需要说明一下:如果 `shard_size` 属性值设置为小于 `size` 属性值,那么 Elasticsearch 会将 `shard_size` 的值设置为 `size` 的值。



注意 请记住,从 Elasticsearch 1.2.0 版本开始,如果将 `shard_size` 或 `size` 属性设置为 0,系统会自动用 `Integer.MAX_VALUE` 替换。

2. 后台数据集过滤

前面提到过, `significant_terms` 聚合中后台数据集指的是整个索引(可能有多个索引)。当然可以通过设置过滤条件(设置 `background_filter` 属性)来缩小后台数据集的大小。如果我们想在指定上下文中查找重要词项,这种过滤是非常有用的。

例如,也许我们想缩小第 1 个例子中后台数据集的范围到真实得分,而不是人为打分,此时可对查询使用 `term` 过滤器:

```
curl -XGET 'localhost:9200/interns/_search?pretty&size=0' -d '{
  "query" : {
    "match" : {
      "intern" : "Richard"
    }
  },
  "aggregations" : {
    "description" : {
      "significant_terms" : {
        "field": "grade",
        "background_filter" : {
          "term" : {
            "type" : "grade"
          }
        }
      }
    }
  }
}
```

进一步查看返回结果,读者可以发现 Elasticsearch 为更小的文档集找出了相关的重要词项:


```
{
  "took" : 4,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 5,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "description" : {
      "doc_count" : 5,
      "buckets" : [ {
        "key" : "bad",
        "doc_count" : 3,
        "score" : 1.02,
        "bg_count" : 2
      } ]
    }
  }
}
```

请注意，这里 `bg_count` 的值由最初范例中的 3 变为了 2。这是因为仅有两个文档的 `grade` 字段中出现了 `bad`，这与我们之前设置的 `background_filter` 的过滤效果吻合。

3. 最小文档数量

对于 `significant_terms` 聚合，一个比较好的想法是能控制重要词项的最小命中文档数，并且这些文档被组织为一个 `bucket`。可以通过设置 `min_doc_count` 属性为任意预期数字来实现该目的。

例如，我们想为计算实习生典型评分的查询添加这个参数。可以将 `min_doc_count` 的默认值降低至 2。修改后的查询如下所示：

```
curl -XGET 'localhost:9200/interns/_search?size=0&pretty' -d '{
  "aggregations" : {
    "grades" : {
      "terms" : {
        "field" : "intern"
      },
      "aggregations" : {
        "significantGrades" : {
          "significant_terms" : {
            "field" : "grade",
```

```

        "min_doc_count" : 2
      }
    }
  }
}
}'

```

上面这个查询的返回结果如下所示：

```

{
  "took" : 3,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 12,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "grades" : {
      "doc_count_error_upper_bound" : 0,
      "sum_other_doc_count" : 0,
      "buckets" : [ {
        "key" : "ralf",
        "doc_count" : 5,
        "significantGrades" : {
          "doc_count" : 5,
          "buckets" : [ {
            "key" : "perfect",
            "doc_count" : 2,
            "score" : 0.32000000000000001,
            "bg_count" : 2
          }, {
            "key" : "good",
            "doc_count" : 3,
            "score" : 0.21000000000000002,
            "bg_count" : 4
          } ]
        }
      } ], {
        "key" : "richard",
        "doc_count" : 5,
        "significantGrades" : {
          "doc_count" : 5,
          "buckets" : [ {
            "key" : "bad",
            "doc_count" : 3,

```

```

        "score" : 0.6,
        "bg_count" : 3
      } ]
    }
  }, {
    "key" : "monica",
    "doc_count" : 2,
    "significantGrades" : {
      "doc_count" : 2,
      "buckets" : [ {
        "key" : "medium",
        "doc_count" : 2,
        "score" : 1.0,
        "bg_count" : 3
      } ]
    }
  } ]
}
}
}
}
}

```

读者可以看到，查询结果与最初的查询有差异，这是因为重要词项的约束条件变弱了。也可以这么理解，检索结果质量现在可能变得更差了。如果将该参数设置为 1，结果可能会很差，如拼写错误的词项，冷僻的词项都会涌现出来，因此不建议这么设置。

有件事情需要注意，当使用 `min_doc_count` 属性时，在聚合计算的第 1 阶段，Elasticsearch 会收集每个 shard 中得分最高的词项。然而，每个 shard 并不知道全局的词频信息，因此会基于 shard 局部词频来推荐候选的重要词项。而 `min_doc_count` 属性在聚合计算的最后阶段才被使用到，此时已经对各 shard 的返回结果进行完合并了。由于这个原因，本应属于重要词项列表中的高频词项被高得分词项取代了。为了避免这种情况，可调大 `shard_size` 属性值，但是此时要承受更大的内存和网络开销。

4. 执行模式

当计算 `significant_terms` 聚合时，可设置不同的执行模式。根据使用者需要，可以将 `execution_hint` 属性设置为 `map` 或 `ordinal`。前者告诉 Elasticsearch 对每个 bucket 中的数据，根据其值进行聚合，而后者则根据初始值进行聚合。在大多数情况下，`execution_hint` 被设置为 `ordinal`，这种模式执行速度稍快，但是计算被强制在初始值上。如果进行 `significant_terms` 聚合的字段为高基字段（该字段有大量不同的词项），那么使用 `map` 模式会更合适一些。



注意 请记住，如果 `execution_hint` 属性不可用，Elasticsearch 会自动忽略之。

5. 更多选项

由于 Elasticsearch 处于持续开发升级过程中，本书中不对可配置属性做一一列举。我

们也忽略了那些很少被使用的属性，这样可以为更常用的选项或特性腾出篇幅。完整的选项列表可参考：<http://www.Elasticsearch.org/guide/en/Elasticsearch/reference/current/search-aggregations-bucket-significantterms-aggregation.html>。

3.3.5 使用限制

在本书撰写之时，`significant_terms` 聚合的使用还存在各种限制。当然，我们还是会使用这种聚合类型，知道其局限之处非常重要。

1. 内存消耗

由于 `significant_terms` 聚合计算针对的是索引字段，因此在计算时会加载所有相关词项到内存中。因此在对大数据集的索引字段进行这种聚合计算时，需要特别小心。此外，也不能通过使用 `doc values` 字段来减少内存使用量，因为 `significant_terms` 聚合不支持。

2. 不应作为顶级聚合使用

当在 `match_all` 查询中使用 `significant_terms` 聚合时，不应将其作为顶级聚合来使用，因为 `match_all` 的作用是返回所有文档。此时前台数据集和后台数据是完全一样的，因此系统感知不到词项在两者中的频率差异，因此无法通过比较词频来发现重要词项。

3. 计数是近似值

Elasticsearch 在统计有多少个文档中包含了某词项时，是基于各 `shard` 的返回信息来估算的。读者需警惕，这种计数很多时候对用户存在误导（例如 `shard` 返回文档中靠前部分文档包含指定词项较少，会导致计数预估偏低）。官方文档曾经指出过，这么设计的动机是为了提高计算性能，当然，随之而来的是计数结果存在一定的不确定性。

4. 不能使用浮点数字段

最后需要注意的一点是，`significant_terms` 聚合计算的目标不能是基于浮点数的字段，可使用基于 `long` 或 `integer` 类型的字段。

3.4 文档分组

Elasticsearch 有很多令人着迷的功能，其中之一就是文档分组（`document folding` 或 `document grouping`）。这也是 Elasticsearch 开发者社区最热门的话题。这并不令人觉得奇怪。因为用户经常需要根据某个值对文档进行分组，尤其是当返回结果数非常大时，有了文档分组功能，实现该目标变得非常容易了。类似的案例中，并不需要向用户展示所有文档，而是从每个分组中返回一个或一些文档。举个例子，也许我们想在图书馆中查询有关

野生动植物的书，返回结果按日期排序，但是限制每年出版的书仅能返回两本。再举个例子，计算某个字段中不同值的个数并显示它们，这种例子很常见，比如说某本书有多个版本，然而我们只想返回其中的一本。

3.4.1 top_hits 聚合

自 Elasticsearch 1.3 版本开始，引入了 top_hits 聚合，这与脚本 (scripting) 的变化有关。在本章后续小节将会介绍脚本。令人着迷的是，我们可以利用这种类型的聚合提供分组功能。事实上，文档分组只是 top_hits 聚合的副产品，我们只能利用 top_hits 聚合来完成这件事。在本节中，我们将只关注 top_hits 聚合，并且假定读者已经对 Elasticsearch 的聚合机制有基本的了解。

如果读者对聚合功能一无所知，可参考本社另外一本著作《Elasticsearch Server, Second Edition》，或参考官方文档：<http://www.Elasticsearch.org/guide/en/Elasticsearch/reference/current/searchaggregations.html>。

这种聚合背后的原理非常简单。每个文档被指派给一个特定的 bucket，系统也能记住这种指派关系。默认情况下，系统只记忆了每个 bucket 中的 3 个文档。现在，继续使用 library 索引来演示我们的案例。

3.4.2 一个例子

为了演示如何使用 top_hits 聚合，我们决定采用下面这个查询：

```
curl -XGET "http://127.0.0.1:9200/library/_search?pretty" -d'
```

```
{
  "size": 0,
  "aggs": {
    "when": {
      "histogram": {
        "field": "year",
        "interval": 100
      },
      "aggs": {
        "book": {
          "top_hits": {
            "_source": {
              "include": [
                "title",
                "available"
              ]
            }
          }
        }
      }
    }
  }
}
```

```

        "size": 1
      }
    }
  }
}
}'

```

在上面的例子中，对年份区间使用了 histogram 聚合，为每 100 年的区间构造了一个 bucket。嵌套的 top_hits 聚合会记住每个 bucket 中得分最高的那个文档（因为这里 size 属性值被设置为 1 了）。我们在查询中添加了 include 选项，主要是为了简化返回结果，这里我们限制只返回被聚合文档的 title 和 available 字段。这意味着 Elasticsearch 的返回结果会与下面类似：

```

{
  "took": 2,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 4,
    "max_score": 0,
    "hits": []
  },
  "aggregations": {
    "when": {
      "buckets": [
        {
          "key_as_string": "1800",
          "key": 1800,
          "doc_count": 1,
          "book": {
            "hits": {
              "total": 1,
              "max_score": 1,
              "hits": [
                {
                  "_index": "library",
                  "_type": "book",
                  "_id": "4",
                  "_score": 1,
                  "_source": {
                    "title": "Crime and Punishment",
                    "available": true
                  }
                }
              ]
            }
          }
        }
      ]
    }
  }
}

```

```

    }
  },
  {
    "key_as_string": "1900",
    "key": 1900,
    "doc_count": 3,
    "book": {
      "hits": {
        "total": 3,
        "max_score": 1,
        "hits": [
          {
            "_index": "library",
            "_type": "book",
            "_id": "3",
            "_score": 1,
            "_source": {
              "title": "The Complete Sherlock
              Holmes",
              "available": false
            }
          }
        ]
      }
    }
  }
]
}

```

查询返回结果中我们感兴趣的部分已经用粗体标出。我们可以看到，由于使用了 `top_hits` 聚合，得分最高的那些文档（每个 bucket 的）被返回了。在这个具体的例子中，查询类型为 `match_all`，因此所有文档的得分都一样，因此为每个 bucket 挑选出得分最高的文档变成很随机的行为了。默认情况下，在 Elasticsearch 中使用 `match_all` 查询与不指定查询效果是等价的。如果用户想采用自定义排序，也完全没有问题。例如，比如你想返回给定世纪的第一本书，我们只需设置适当的排序选项，就像下面这个查询一样：

```

curl -XGET 'http://127.0.0.1:9200/library/_search?pretty' -d '{
  "size": 0,
  "aggs": {
    "when": {
      "histogram": {
        "field": "year",
        "interval": 100
      },
    },
  },
}

```

```

"aggs": {
  "book": {
    "top_hits": {
      "sort": {
        "year": "asc"
      },
      "_source": {
        "include": [
          "title",
          "available"
        ]
      },
      "size": 1
    }
  }
}
}
}
}'

```

请注意上面这个查询中的粗体部分。我们为 `top_hits` 聚合添加了排序选项，因此返回结果会基于 `year` 字段排序。这意味着每个 `bucket` 里年份最小的那本书将会被返回。

额外参数

除了排序及指定排序字段以外，`top_hits` 聚合还有一些其他功能。Elasticsearch 支持其他若干种文档检索功能。但是在这里不一一赘述，因为读者如果了解聚合功能，那么基本上已经对这些功能非常熟悉了。然而，为了支持本章的主题，下面再补充一个例子：

```

curl -XGET 'http://127.0.0.1:9200/library/_search?pretty' -d '{
  "query": {
    "filtered": {
      "query": {
        "match": {
          "all": "quiet"
        }
      },
      "filter": {
        "term": {
          "copies": 1,
          "_name": "copies_filter"
        }
      }
    }
  }
}'

```



```

    }
  },
  "size": 0,
  "aggs": {
    "when": {
      "histogram": {
        "field": "year",
        "interval": 100
      },
      "aggs": {
        "book": {
          "top_hits": {
            "highlight": {
              "fields": {
                "title": {}
              }
            },
            "explain": true,
            "version": true,
            "_source": {
              "include": [
                "title",
                "available"
              ]
            },
            "fielddata_fields" : ["title"],
            "script_fields": {
              "century": {
                "script": "(doc[\"year\"].value /
                100).intValue()"
              }
            },
            "size": 1
          }
        }
      }
    }
  }
}

```

读者可以发现，我们的查询包含了如下功能：

- ❑ 指定过滤器和查询（例子中为 `copies_filter`）
- ❑ 获取文档版本信息

- ❑ 文档源过滤（选择返回哪些字段）
- ❑ 使用 field-data 字段及脚本字段
- ❑ 获取文档被召回的解释性信息
- ❑ 高亮显示

3.5 文档关系

当 Elasticsearch 受到越来越多的关注时，它慢慢演化，变得不仅仅是一个搜索引擎了。它可以被视为一种数据分析解决方案，或者一种数据存储系统。既能存储数据又能快速检索高效的全文检索，看起来是个好主意。Elasticsearch 不仅能存储文档，还能进行全文检索，对数据进行分析赋予其语义，这是传统 SQL 数据库所不能企及的。不过，如果您对传统关系数据库非常了解，在使用 Elasticsearch 时就会理解对文档关系建模之必要性。不幸的是，在 Elasticsearch 做到这点不太容易，关系数据库中的很多特性 Elasticsearch 并不具备，因为其底层使用的是倒排索引，与传统数据库实现机制不同。但是利用嵌套对象和 parent-child 功能，Elasticsearch 可以提供文档建模功能（可参考《Elasticsearch Server, Second Edition》）。现在来看看文档建模功能及其陷阱。

3.5.1 对象类型

Elasticsearch 对数据建模和全文索引构建的限制少之又少。不像关系数据库，Elasticsearch 能很自如地索引结构化对象。这意味着即便是对 JSON 文档的索引，也完全不在话下。请查看下面这个文档：

```
{
  "title": "Title",
  "quantity": 100,
  "edition": {
    "isbn": "1234567890",
    "circulation": 50000
  }
}
```

读者可以看到，上面的文档只有两个简单的字段及一个嵌套对象（edition 对象）及其属性。范例中用到的 mapping 也很简单（保存在 relations.json 文件中），如下所示：

```
{
  "book" : {
    "properties" : {
      "title" : { "type": "string" },
      "quantity" : { "type": "integer" },
```

```

    "edition" : {
      "type" : "object",
      "properties" : {
        "isbn" : { "type" : "string", "index" : "not_analyzed" },
        "circulation" : { "type" : "integer" }
      }
    }
  }
}

```

不幸的是，如果想要一切工作正常，内部对象与其父对象必须是一对一关系。例如添加第 2 个对象，如下所示：

```

{
  "title": "Title",
  "quantity": 100,
  "edition": [
    {
      "isbn": "1234567890",
      "circulation": 50000
    },
    {
      "isbn": "9876543210",
      "circulation": 2000
    }
  ]
}

```

Elasticsearch 会把内部对象打平（flatten）。前面的那个文档会变得与下面这个文档类似（当然，`_source` 字段会保持不变）：

```

{
  "title": "Title",
  "quantity": 100,
  "edition": {
    "isbn": [ "1234567890", "9876543210" ],
    "circulation": [50000, 2000 ]
  }
}

```

这并不是我们所预期的，这种文档表示会导致问题，比如说当我们想查找包含指定 ISBN 及发行量的图书，而此时 Elasticsearch 会返回包含指定 ISBN 但是任意发行量的图书。

可使用下面的命令索引文档并测试查询效果，索引命令如下：

```

curl -XPOST 'localhost:9200/object/doc/1' -d '{
  "title": "Title",
  "quantity": 100,
  "edition": [

```

```
{
  "isbn": "1234567890",
  "circulation": 50000
},
{
  "isbn": "9876543210",
  "circulation": 2000
}
]
}'
```

现在可以来执行一个查询，如果搜索那些 isbn 字段值为 1234567890 且 circulation 字段值为 2000 的图书，将不会返回任何文档。然而，可执行下面这个查询：

```
curl -XGET 'localhost:9200/object/_search?pretty' -d '{
  "fields" : [ "_id", "title" ],
  "query" : {
    "bool" : {
      "must" : [
        {
          "term" : {
            "isbn" : "1234567890"
          }
        },
        {
          "term" : {
            "circulation" : 2000
          }
        }
      ]
    }
  }
}'
```

下面是 Elasticsearch 返回的结果：

```
{
  "took" : 5,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
```



```

    "total" : 1,
    "max_score" : 1.0122644,
    "hits" : [ {
      "_index" : "object",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 1.0122644,
      "fields" : {
        "title" : [ "Title" ]
      }
    } ]
  }
}

```

可以通过重新排列 mapping 和文档来避免交叉查找，所以源文档看起来会像下面这样：

```

{
  "title": "Title",
  "quantity": 100,
  "edition": {
    "isbn": ["1234567890", "9876543210"],
    "circulation_1234567890": 50000,
    "circulation_9876543210": 2000
  }
}

```

现在可以使用前面提到的那个查询了，此时能利用字段之间的关联关系，但是代价是会构建出更复杂的查询。此时会引发更重要的问题，mapping 中将会包含字段中所有数值的信息，如果文档字段中包含超过两个值，结果可能不是我们想要的。换个角度来说，这里并不允许我们构建某些复杂的查询，例如查找所有销量大于 10000 且 ISBN 以 23 开头的图书。这种查询，嵌套对象是更好的解决方案。

总结一下，object 类型只有在很简单的情景中好用，如不存在跨字段查找等麻烦时，即你不需要在嵌套对象中搜索，或者仅需要在单个字段中搜索而不需要关联多个字段时。

3.5.2 嵌套文档

从 mapping 的角度来看，定义一个嵌套文档很简单，仅仅是把之前的 object 替换为 nested 类型（object 是 Elasticsearch 默认类型）。举个例子，我们将前面的范例修改一下，让它使用嵌套文档：

```

{
  "book" : {
    "properties" : {
      "title" : { "type": "string" },
      "quantity" : { "type": "integer" },
      "edition" : {
        "type" : "nested",

```

```
"properties" : {  
  "isbn" : { "type" : "string", "index" : "not_analyzed" },  
  "circulation" : { "type" : "integer" }  
}  
}  
}  
}
```

当使用嵌套文档时，Elasticsearch 实际上是为主对象创建了一个文档（这里也可以称之为父对象，但是考虑到避免与后面将要介绍的 parent-child 功能混淆，所以把它叫作主对象），并为内部对象创建额外的文档。普通查询中，这些额外文档被自动过滤掉，不会被搜索到或展示出来。这在 Apache Lucene 中被称为块连接（block join）（块连接详情可参考 Lucene 委员会成员 Mike McCandless 的博客：<http://blog.mikemccandless.com/2012/01/searching-relational-content-with.html>）。出于性能方面的考虑，额外的文档与主文档保存在一个索引块（segment block）中。

这也是为什么嵌套文档必须要与主文档同时被索引。因为相互关联的两端文档的存储与索引是同时进行的。因此有人也将嵌套文档称为索引期连接（index-time join）。当文档都很小且主文档数据易于获取时，这种强关联关系并不会造成什么问题。如果这些文档很大，关联双方之一变化较频繁，那么重建另外一部分文档变得不太现实了。另外就是如果一个嵌套文档属于多个主文档时，问题会变得非常棘手。而这些问题在 parent-child 功能面前会迎刃而解。

回到前面的例子中，对我们的索引做些改变，转而使用嵌套对象（nested object），并将我们的查询修改为嵌套查询（nested query）。此时查询不会返回任何文档，这是因为单个嵌套文档并不与这样的查询匹配。

3.5.3 parent-child 关系

谈及 parent-child 功能，应从其最大优势谈起——关系两端的文档是相互独立的——每端的文档都是独立索引的。这么做也是有代价的，会导致更复杂的查询及更慢的查询性能。Elasticsearch 中提供了特殊的查询和过滤器来处理这种关系。parent-child 关系又被称为查询期连接（query-time join）。parent-child 关系的第二个优势是这种关系适用于大型应用及多节点场景，这个优势看起来更有价值。现在让我们来看看如何在多节点分布式 Elasticsearch 集群中使用 parent-child 关系。



注意 读者请记住这里与嵌套文档的不同之处，此时子文档检索并不强制在主文档上下文中进行，而这是嵌套文档机制所不能做到的。

集群中的 parent-child 关系

为了更好地演示集群中 parent-child 关系用法，让我们来创建两个索引：rel_pch_m 索引和 rel_pch_s 索引，前者存储主文档，后者存储子文档。索引创建命令如下：

```
curl -XPUT localhost:9200/rel_pch_m -d '{ "settings" : {  
  "number_of_replicas" : 0 } }'  
curl -XPUT localhost:9200/rel_pch_s -d '{ "settings" : {  
  "number_of_replicas" : 0 } }'
```

索引 rel_pch_m 对应的 mapping 很简单，可发送下面的命令至 Elasticsearch 来设置它：

```
curl -XPOST localhost:9200/rel_pch_m/book/_mapping?pretty -d '{  
  "book" : {  
    "properties" : {  
      "title" : { "type": "string" },  
      "quantity" : { "type": "integer" }  
    }  
  }  
'
```

索引 rel_pch_s 对应的 mapping 同样也很简单，只是需要通知 Elasticsearch 其父文档类型。可使用下面的命令将第二个索引的 mapping 发送给 Elasticsearch：

```
curl -XPOST localhost:9200/rel_pch_s/edition/_mapping?pretty -d '{  
  "edition" : {  
    "_parent" : {  
      "type" : "book"  
    },  
    "properties" : {  
      "isbn" : { "type" : "string", "index" : "not_analyzed" },  
      "circulation" : { "type" : "integer" }  
    }  
  }  
'
```

最后一步是向索引中导入数据。不妨先索引 10000 个文档。下面是文档范例：

```
{ "index": { "_index": "rel_pch_m", "_type": "book", "_id": "1" } }  
{ "title": "Doc no 1", "quantity": 101 }  
{ "index": { "_index": "rel_pch_s", "_type": "edition", "_id": "1",  
  "_parent": "1" } }  
{ "isbn": "no1", "circulation": 501 }
```



注意 如果读者很好奇并想自己做实验，可以找到随书的 bash 脚本 create_relation_indices.sh，通过运行它来生成范例数据。

实验很简单，每个种类型（book 和 edition 类型）文档各 10000 个。关键是 `_parent` 字段。在我们的范例中，该字段值总是被设置为 1。于是索引数据中有 10000 本书，又有 10000 个版本，每个版本属于特定的一本书。我们的范例比较极端，但是却揭示了一个重要的事实。



注意 如果想要做些可视化展示，可使用 ElasticHQ 插件，详情请登录 <http://www.elastichq.org>。

首先看看关系中的父文档部分，可参考下面这个截图：

The screenshot shows the 'Shards' tab for the index 'rel_pch_m'. It displays a table with 5 shards, all in a 'STARTED' state. Shards 0, 1, and 2 are on node 'Samuel Silke', while shards 3 and 4 are on node 'Stygorr'. Each shard contains approximately 2,000 documents.

Shard	State	# Docs	Size	Primary?	Node
0	STARTED	2,000	137.4KB	true	Samuel Silke
1	STARTED	1,999	137.3KB	true	Samuel Silke
2	STARTED	2,000	137.3KB	true	Cat-Man
3	STARTED	2,001	137.5KB	true	Stygorr
4	STARTED	2,000	137.3KB	true	Samuel Silke

如读者所见，索引的 5 个 shard 分布在 3 个不同的节点上。每个 shard 上的文档数类似。这正是我们所期待的，Elasticsearch 使用哈希算法来确定每个文档应放置在哪个 shard 上。

现在再来看看第二个索引，该索引中保存的是子文档，详情参考下面这个截图：

The screenshot shows the 'Shards' tab for the index 'rel_pch_s'. It displays a table with 5 shards, all in a 'STARTED' state. Shards 0, 1, 3, and 4 are on node 'Samuel Silke', while shard 2 is on node 'Cat-Man' and shard 4 is on 'Stygorr'. Shards 0, 1, 3, and 4 are empty (0 docs), while shard 2 contains 10,000 documents.

Shard	State	# Docs	Size	Primary?	Node
0	STARTED	0	123.0B	true	Samuel Silke
1	STARTED	0	123.0B	true	Cat-Man
2	STARTED	10,000	594.7KB	true	Samuel Silke
3	STARTED	0	123.0B	true	Stygorr
4	STARTED	0	123.0B	true	Samuel Silke

两个索引的情况有所不同。此时仍然有 5 个 shard，但是其中 4 个为空索引，只有最后一个 shard 中索引了 10000 个文档。因此肯定是哪里出问题了，所有的文档都被索引在一个 shard 上了，这不是我们所预期的。Elasticsearch 总是将父文档相同的文档放置在同一个 shard 中（换句话说，子文档的 `routing` 参数值总是与其 `parent` 参数值相等）。在我们的范例中，如果某些父文档有多个子文档，会导致文档在 shard 之间的不均匀分布，这会引发性能和存储问题。比如说某些 shard 空闲，而某些 shard 超负荷。

3.5.4 其他解决方案

读者已经了解到了，使用 Elasticsearch 处理文档关系会有这样那样的问题。Elasticsearch 的最大价值在于全文检索和数据分析，而不是文档关系建模。如果您的应用对文档关系建模要求非常高，或者全文检索并不是应用的核心功能，那么可以考虑使用带全文检索扩展的 SQL 数据库。如果这些全文检索扩展不如 Elasticsearch 那么灵活或高性能，也不用诧异，毕竟它们的主业不是全文检索，更重要的是我们得到了强大的关系数据处理支持。不过，在大多数案例中，改变数据架构及通过反范式 (de-normalization) 设计等手段消除关系就足以应付应用需求了。

3.6 Elasticsearch各版本中脚本的变化

脚本 (scripting) 是 Elasticsearch 提供的最强悍的功能之一。可使用脚本进行计算分值、文本相关性、数据过滤、数据分析。尽管脚本在很多情况中会导致较低的性能，如为每个文档计算得分，但是我们认为 Elasticsearch 提供的这种功能是非常重要的。本节中将会介绍脚本功能在各版本中的变化，也是对《Elasticsearch Server, Second Edition》一书相关章节的补充。

3.6.1 脚本变迁

Elasticsearch 中的脚本功能自 1.0 版本以来重构过若干次。于是很多用户困惑不已，为什么之前可用的脚本在升级到 1.2 版本以后变得不可用了，这是非常常见的情形。本节将会介绍这些变化。

1. 安全事项

在 Elasticsearch 1.1 版本的生命周期中，发现了一个重大安全隐患（参考 <http://bouk.co/blog/Elasticsearch-rce/>）：Elasticsearch 默认配置并不安全。因此在 Elasticsearch 1.2 版本中，动态脚本功能被默认禁用了。这使得使用 Elasticsearch 更安全，但是脚本的使用却变得更复杂了。

2. Groovy——新一代默认脚本语言

自 Elasticsearch 1.3 版本起，使用 Groovy 作为默认的脚本语言（Groovy 详情可参考 <http://groovy.codehaus.org/>）。使用 Groovy 的理由很简单，因为它能被控制在自己的沙箱 (sandbox) 中，能防止动态脚本做对集群或操作系统有害的事情。因为 Groovy 能被沙箱化，所以可以通过它来使用动态脚本。换句话说，从 Elasticsearch 1.3 版本以后，如果一种脚本语言能被沙箱化，它就可以在动态脚本中被使用。然而需要注意的是，Groovy 并不

能做好所有的事情：自 Elasticsearch 1.3 起，允许用户使用 Lucene 表达式（本节中将会介绍）。然而从 1.3.8 版本及 1.4.3 版本发布后，动态脚本即便是 Groovy 也默认被禁用。因此，如果想在 Groovy 中使用动态脚本，需要在 Elasticsearch.yml 中添加 `script.groovy.sandbox.enabled` 属性，并将其值设置为 `true`，或者设置 Elasticsearch 预存储脚本代码实现有限的动态化。但是读者需谨记，开启动态脚本功能会对外暴露安全隐患，使用时需谨慎。

3. MVEL 语言的移除

由于安全问题及 Groovy 的引入，从 Elasticsearch 1.4 版本开始，MVEL 默认不可用。默认脚本语言为 Groovy，如果想使用 MVEL，则需使用相应的插件。请记住，如果您想移除 MVEL 而转用 Groovy，是很容易的，甚至可以安装 MVEL 插件，但是禁止动态执行 MVEL 脚本。

3.6.2 Groovy 简单介绍

Groovy 是一种基于 Java 虚拟机的动态语言。它构建在 Java 之上，同时又具备 Python、Ruby、Smalltalk 等语言的多种优点。Groovy 的话题远远超出了本书的范围，我们只是简单介绍一下，因为从 Elasticsearch 1.4 起，Groovy 成为默认的脚本语言。如果读者对 Elasticsearch 中的 Groovy 使用已经有所了解，可跳过本小节，阅读后续的 3.6.3 节。



读者需谨记，Groovy 仅在 1.3.8 版本和 1.4.3 版本前可沙箱化。而此后 Groovy 动态脚本默认是禁用的，除非在 Elasticsearch 中配置使用它。后面我们的范例查询涉及动态脚本，需要在 Elasticsearch.yml 配置文件中设置 `script.groovy.sandbox.enabled` 属性值为 `true`。

1. 将 Groovy 当成脚本语言来使用

在介绍 Groovy 之前，先来了解如何在 Elasticsearch 中使用脚本。在此之前，请检查您使用的 Elasticsearch 的版本。如果版本老于 1.4，则需要在 `query` 中添加 `lang` 属性，其值设置为 `groovy`，如下所示：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "fields" : [ "_id", "_score", "title" ],
  "query" : {
    "function_score" : {
      "query" : {
        "match_all" : {}
      },
      "script_score" : {
        "lang" : "groovy",
        "script" : "_index[\"title\"].docCount() "
```

```

    }
  }
}
}'

```

如果使用的版本为 1.4 或者更新版本，则不需要设置。因为此时 Elasticsearch 默认使用 Groovy 作为脚本语言。

2. 脚本中的变量定义

Elasticsearch 允许 Groovy 在脚本中定义变量。为了定义新变量，可使用 `def` 关键字，后面紧随变量名及变量值。例如，我们想声明一个叫 `sum` 的变量，并将它初始化为 0，可使用下面这段代码：

```
def sum = 0
```

当然，不限于上面这样的变量定义，也可用一系列值来初始化变量，如用 4 个值：

```
def listOfValues = [0, 1, 2, 3]
```

还可用一个数值区间来定义变量，如下面这行代码，数值区间为 0 ~ 9：

```
def rangeOfValues = 0..9
```

最后，也可以定义字典：

```
def map = ['count':1, 'price':10, 'quantity': 12]
```

上面这行代码执行结果是定义了一个字典，里面有个 3 个 keys (`count`, `price`, `quantity`)，其值分别为 1、10、12。

3. 条件语句

当然也可以在脚本中定义条件语句。例如定义标准的 `if-else if-else` 结构：

```

if (count > 1) {
    return count
} else if (count == 1) {
    return 1
} else {
    return 0
}

```

也可使用经典的三元操作符：

```
def isHigherThanZero = (count > 0) ? true : false
```

上面的代码中，根据 `count` 的值是否大于 0，将变量 `isHigherThanZero` 的值设置为 `true` 或者 `false`。

该语言当然也支持标准的 `switch` 结构，运行用户以很原始的方式基于变量值检测来执行具体的语句。范例如下所示：

```
def isEqualToTenOrEleven = false;
switch (count) {
    case 10:
        isEqualToTenOrEleven = true
        break
    case 11:
        isEqualToTenOrEleven = true
        break
    default:
        isEqualToTenOrEleven = false
}
```

上面的代码会将变量 `isEqualToTenOrEleven` 的值设置为 `true`，如果 `count` 的值等于 10 或 11，否则将 `isEqualToTenOrEleven` 的值设置为 `false`。

4. 循环语句

在 Elasticsearch 中的 Groovy 脚本中当然也可以使用循环语句。下面我们以 `while` 循环为例，当括号中条件为 `true` 时，循环体中的代码将一直执行下去：

```
def i = 2
def sum = 0
while (i > 0) {
    sum = sum + i
    i--
}
```

上面的循环被执行两次然后退出。第 1 趟循环中，变量 `i` 的值为 2，此时条件 `i>0` 为 `true`，因此执行循环体中代码。第 2 趟循环中变量 `i` 的值为 1，此时条件 `i>0` 依然为 `true`，因此也执行循环体中代码。第 3 趟循环中，`i` 的值为 0，此时括号内 `i>0` 为 `false`，此时循环结束。

也可以使用 `for` 循环，如果读者之前有过编程经验，会很容易理解 `for` 循环。例如，使用 `for` 循环执行 10 次。可参考下面的代码：

```
def sum = 0
for ( i = 0; i < 10; i++) {
    sum += i
}
```

也可以使用 `for` 循环在一个数值区间上迭代：

```
def sum = 0
for ( i in 0..9 ) {
    sum += i
}
```

或者在一个数值列表上迭代：

```
def sum = 0
for ( i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] ) {
    sum += i
}
```


如果我们有一个字典，也可以通过字典的 entry 来迭代：

```
def map = ['quantity':2, 'value':1, 'count':3]
def sum = 0
for ( entry in map ) {
    sum += entry.value
}
```

5. 一个例子

现在可以开始介绍 Groovy 了，现在我们来执行一个脚本以修改文档的得分。将在脚本中实现如下评分算法：

- ❑ 如果 year 字段的值大于 800，那么文档 (book) 得分为 1.0。
- ❑ 如果 year 字段的值为 1800 ~ 1900，那么文档得分为 2.0。
- ❑ 其余文档得分为 year 字段值减 1000。

下面的查询用来实现前面的评分算法：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "fields" : [ "_id", "_score", "title", "year" ],
  "query" : {
    "function_score" : {
      "query" : {
        "match_all" : {}
      },
      "script_score" : {
        "lang" : "groovy",
        "script" : "def year = doc[\"year\"].value; if (year < 1800) {
          return 1.0 } else if (year < 1900) { return 2.0 } else { return
          year - 1000 }"
      }
    }
  }
}
```



注意 读者可能注意到了，在语句 `def year = doc["year"].value` 后面，我们用分号分隔了，这是为了告诉 Groovy，只是一行代码，分号前是赋值语句，分号后是另一条语句。

上面的查询的返回结果如下所示：

```
{
  "took" : 4,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  }
}
```

```

},
"hits" : {
  "total" : 6,
  "max_score" : 961.0,
  "hits" : [ {
    "_index" : "library",
    "_type" : "book",
    "_id" : "2",
    "_score" : 961.0,
    "fields" : {
      "title" : [ "Catch-22" ],
      "year" : [ 1961 ],
      "_id" : "2"
    }
  }, {
    "_index" : "library",
    "_type" : "book",
    "_id" : "3",
    "_score" : 936.0,
    "fields" : {
      "title" : [ "The Complete Sherlock Holmes" ],
      "year" : [ 1936 ],
      "_id" : "3"
    }
  }, {
    "_index" : "library",
    "_type" : "book",
    "_id" : "1",
    "_score" : 929.0,
    "fields" : {
      "title" : [ "All Quiet on the Western Front" ],
      "year" : [ 1929 ],
      "_id" : "1"
    }
  }, {
    "_index" : "library",
    "_type" : "book",
    "_id" : "6",
    "_score" : 904.0,
    "fields" : {
      "title" : [ "The Peasants" ],
      "year" : [ 1904 ],
      "_id" : "6"
    }
  }, {
    "_index" : "library",
    "_type" : "book",
    "_id" : "4",
    "_score" : 2.0,
    "fields" : {
      "title" : [ "Crime and Punishment" ],

```

```

        "year" : [ 1886 ],
        "_id" : "4"
    }, {
        "_index" : "library",
        "_type" : "book",
        "_id" : "5",
        "_score" : 1.0,
        "fields" : {
            "title" : [ "The Sorrows of Young Werther" ],
            "year" : [ 1774 ],
            "_id" : "5"
        }
    }
]
}

```

如我们预期的那样，脚本完成了它的使命。

6. 额外的说明

显然，前面介绍的 Groovy 的知识并不是一份全面的指南，我们也无意做这方面的努力。因为 Groovy 的全面介绍远远超出了本书的话题，本书为领读者稍做介绍，了解我们能够通过 Groovy 获得什么。如果读者对 Groovy 有强烈的兴趣，想进一步了解，建议登录其官网阅读相关文档：<http://groovy.codehaus.org/>。

3.6.3 全文检索中的脚本

前面介绍了利用文档中数据计算得分的例子，而在全文检索上下文中，可以在脚本中使用全文检索统计量，如词频、文档频率等。下面可以看看有哪些可用的信息。

1. 字段相关信息

首先能想到的脚本中可以使用的文本相关信息就是字段相关信息。Elasticsearch 中可使用的字段相关信息包括以下这些。

- ❑ `_index['field_name'].docCount()`: 有多少文档包含该字段。该统计量不考虑被删除文档。
- ❑ `_index['field_name'].sumttf()`: 所有文档给定字段中词项出现次数之和。
- ❑ `_index['field_name'].sumdf()`: 文档频率之和。指定字段中词项的文档频率之和。



注意 请记住，上面这些统计量是 shard 内统计量，同一个统计量在不同 shard 之间的值可能不一样。

例如，也许你想将文档得分赋值为所在 shard 内包含 title 字段的文档的个数，可执行下面这个查询：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "fields" : [ "_id", "_score", "title" ],
  "query" : {
    "function_score" : {
      "query" : {
        "match_all" : {}
      },
      "script_score" : {
        "lang" : "groovy",
        "script" : "_index[\"title\"] .docCount()"
      }
    }
  }
}'
```

如果查看返回结果，会类似于下面这样：

```
{
  "took" : 3,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 6,
    "max_score" : 2.0,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "1",
      "_score" : 2.0,
      "fields" : {
        "title" : [ "All Quiet on the Western Front" ],
        "_id" : "1"
      }
    }, {
      "_index" : "library",
      "_type" : "book",
      "_id" : "6",
      "_score" : 2.0,
      "fields" : {
        "title" : [ "The Peasants" ],
        "_id" : "6"
      }
    }, {
      "_index" : "library",
      "_type" : "book",
```



```

    "_id" : "4",
    "_score" : 1.0,
    "fields" : {
      "title" : [ "Crime and Punishment" ],
      "_id" : "4"
    }
  }, {
    "_index" : "library",
    "_type" : "book",
    "_id" : "5",
    "_score" : 1.0,
    "fields" : {
      "title" : [ "The Sorrows of Young Werther" ],
      "_id" : "5"
    }
  }, {
    "_index" : "library",
    "_type" : "book",
    "_id" : "2",
    "_score" : 1.0,
    "fields" : {
      "title" : [ "Catch-22" ],
      "_id" : "2"
    }
  }, {
    "_index" : "library",
    "_type" : "book",
    "_id" : "3",
    "_score" : 1.0,
    "fields" : {
      "title" : [ "The Complete Sherlock Holmes" ],
      "_id" : "3"
    }
  }
]
}

```

如读者所见，查询返回了 6 个文档。前两个文档得分为 2.0，后 4 个文档得分为 1.0，这意味着这两批文档分别位于不同的 shard 中。

2. Shard 级信息

也有很多可使用的 shard 级信息。

- ❑ `_index.numDocs()`: shard 中的文档数。
- ❑ `_index.maxDoc()`: shard 内部最大文档 ID。
- ❑ `_index.numDeletedDocs()`: shard 内已删除文档数。



注意

请记住，上面这些统计量是 shard 内统计量，同一个统计量在不同 shard 之间的值可能不一样。

举个例子，如果我们基于每个 shard 的最大文档 ID 对文档进行排序，可使用下面这个查询：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "fields" : [ "_id", "_score", "title" ],
  "query" : {
    "function_score" : {
      "query" : {
        "match_all" : {}
      },
      "script_score" : {
        "lang" : "groovy",
        "script" : "_index.maxDoc()"
      }
    }
  }
}'
```

当然，像这样简单使用单个统计量并没有什么意义。如果结合其他的全文检索统计量使用，它们的用途就大大增强了。

3. 词项级信息

在脚本中还可以使用另外一种统计量，词项级统计量。Elasticsearch 中有如下词项级统计量。

- `_index['field_name']['term'].df()`：指定字段中，有多少个文档中出现过某词项。
- `_index['field_name']['term'].ttf()`：指定词项在所有文档的指定字段中出现的次数。
- `_index['field_name']['term'].tf()`：某词项在文档的指定字段中出现的次数。

为了向读者演示如何使用这些统计量，可使用下面的命令索引两个文档：

```
curl -XPOST 'localhost:9200/scripts/doc/1' -d '{"name":"This is a
document"}'
curl -XPOST 'localhost:9200/scripts/doc/2' -d '{"name":"This is a
second document after the first document"}'
```

现在，基于某词项在 `name` 字段中出现的次数来过滤文档。例如，我们想找出在 `name` 字段中 `document` 词项至少出现过两次的文档。为实现该目的，可使用下面这个查询：

```
curl -XGET 'localhost:9200/scripts/_search?pretty' -d '{
  "query" : {
    "filtered" : {
      "query" : {
        "match_all" : {}
      },

```

```

    "filter" : {
      "script" : {
        "lang" : "groovy",
        "script": "_index[\"name\"] [\"document\"].tf() > 1"
      }
    }
  }
}
}'

```

查询返回结果如下所示：

```

{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "scripts",
      "_type" : "doc",
      "_id" : "2",
      "_score" : 1.0,
      "_source": {"name": "This is a second document after the first
document"}
    } ]
  }
}

```

如读者所见，Elasticsearch 返回了预期的结果。

4. 更多的高阶词项信息

除了前面提及的那些统计量，还有更多的词项级信息可在脚本中使用，如词项位置、偏移量、负载 (payload)。可使用 `_index['field_name'].get('term', OPTION)` 获取这些统计量，这里的 `OPTION` 是下面这些选项：

- ❑ `_OFFSETS`：词项偏移量
- ❑ `_PAYLOADS`：词项负载
- ❑ `_POSITION`：词项位置



注意

如果想获取位置、偏移量等信息，需要在索引期设置索引字段的处理方式。

除此之外，还可以使用 `_CACHE` 选项，该选项允许我们多次迭代词项所有的位置信息，选项可以用 “|” 操作符组合使用。如果您既想获取 `name` 字段中 `document` 词项的位置信息，又想获取其偏移量信息，可在脚本中使用下面的表达式：

```
_index['title'].get('document', _OFFSETS | _POSITIONS).
```

有一点读者需记住，类似上面这样的组合选项的处理结果因采用的选项而异，通常会包含下面这些信息。

- ❑ `startOffset`: 词项起始偏移量
- ❑ `endOffset`: 词项结束偏移量
- ❑ `payload`: 词项负载
- ❑ `payloadAsInt(value)`: 将词项负载转化为整数值，如果负载缺失用整数 `value` 替代
- ❑ `payloadAsFloat(value)`: 将词项负载转化为浮点值，如果负载缺失用浮点值 `value` 替代
- ❑ `payloadAsString(value)`: 将词项负载转化为字符串，如果负载缺失用字符串 `value` 替代
- ❑ `position`: 词项位置

出于演示用途，我们使用下面的 `mapping` 创建一个新索引：

```
curl -XPOST 'localhost:9200/scripts2' -d '{
  "mappings" : {
    "doc" : {
      "properties" : {
        "name" : { "type" : "string", "index_options" : "offsets" }
      }
    }
  }
}'
```

之后，用下面的命令索引两个文档：

```
curl -XPOST 'localhost:9200/scripts2/doc/1' -d '{"name": "This is the first document"}'
curl -XPOST 'localhost:9200/scripts2/doc/2' -d '{"name": "This is a second simple document"}'
```

现在来设置文档的得分计算方法，将每个文档得分设置为 `name` 字段中 `document` 词项的 `startOffset` 之和。可执行下面这个查询：

```
curl -XGET 'localhost:9200/scripts2/_search?pretty' -d '{
  "query" : {
    "function_score" : {
      "query" : {
        "match_all" : {}
      },

```



```

"script_score" : {
  "lang" : "groovy",
  "script": "def termInfo = _index[\"name\"].get(\"document\",_OFFSETS);
def sum = 0; for (offset in termInfo) { sum += offset.startOffset; };
return sum;"
}
}
}'

```

Elasticsearch 返回结果如下所示:

```

{
  "took" : 3,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 2,
    "max_score" : 24.0,
    "hits" : [ {
      "_index" : "scripts2",
      "_type" : "doc",
      "_id" : "2",
      "_score" : 24.0,
      "_source":{"name":"This is a second simple document"}
    }, {
      "_index" : "scripts2",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 18.0,
      "_source":{"name":"This is the first document"}
    } ]
  }
}

```

读者不难发现,脚本正常工作了。下面是前面脚本代码的更规整的写法:

```

def termInfo = _index['name'].get('document',_OFFSETS);
def sum = 0;
for (offset in termInfo) {
  sum += offset.startOffset;
};
return sum;

```

这段脚本代码其实并不复杂。首先,获取一个对象的偏移量信息;然后,创建一个变量保存偏移量之和;之后,使用循环对偏移量进行累加(同一个词项在多个位置出现了)。

最后的总和作为文档得分返回。



注意

除了前面各节介绍的内容之外，我们也可以使用词项向量信息 (term vector)，词项向量的构建需要在索引期设置。可使用 `_index.termVectors()` 表达式获取词项向量，使用该表达式会返回一个 Lucene 的 Fields 对象实例。更多 Fields 对象细节，可参考官方文档：https://lucene.apache.org/core/4_9_0/core/org/apache/lucene/index/Fields.html。

3.6.4 Lucene 表达式

尽管这项功能被官方标识为实验性的，笔者还是打算花费一定篇幅来介绍它。它是一种崭新的但是又非常有用的特性。Lucene 表达式吸引人的理由是它执行速度非常快，甚至与原生脚本一样快，但是它也像动态脚本那样存在某些限制。本节内容将展示 Lucene 表达式的一些功能。

1. 基础知识

Lucene 支持将 JavaScript 表达式编译成 Java 字节码。这也是 Lucene 表达式的实际工作原理。正因如此，它们和普通的 Elasticsearch 脚本执行得一样快。Lucene 表达式可以在 Elasticsearch 的下面这些功能中被使用：

- ☐ 用于排序的脚本
- ☐ 数值字段中的聚合
- ☐ script_score 查询中的 function_score 中
- ☐ 使用 script_fields 的查询中

除此之外，用户需记住：

- ☐ Lucene 表达式仅能在数值字段上使用
- ☐ Lucene 表达式不能访问存储字段
- ☐ 字段缺失值用数值 0 替换
- ☐ 可使用 `_score` 访问文档得分，可使用 `doc['field_name'].value` 访问文档的单值数值字段中的值
- ☐ Lucene 表达式中不允许使用循环，只能使用单条语句

2. 一个例子

通过前面的介绍，读者已经可以利用 Lucene 表达式来修改文档得分了。我们再回顾一下之前提到过的 library 索引，我们将每个命中文档的得分赋值为其出版年份数的 10%。为实现该目的，可执行下面这个查询：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "fields" : [ "_id", "_score", "title" ],
  "query" : {
    "function_score" : {
      "query" : {
        "match_all" : {}
      },
      "script_score" : {
        "lang" : "expression",
        "script" : "_score + doc[\"year\"].value * percentage",
        "params" : {
          "percentage" : 0.1
        }
      }
    }
  }
}
```

查询本身很简单，我们感兴趣的是查询的结构。首先，用 `function_score` 查询包装了 `match_all` 查询。这是因为我们希望所有文档命中，并且对文档得分进行定制。然后设置脚本语言为表达式（将 `lang` 属性值设置为 `expression`），这么做的目的是通知 Elasticsearch 脚本类型为 Lucene 表达式。当然，我们提供了脚本，也需要提供对应的参数，就像我们使用其他脚本一样。前面的查询将会返回类似下面这样的结果：

```
{
  "took" : 4,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 6,
    "max_score" : 197.1,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "2",
      "_score" : 197.1,
      "fields" : {
        "title" : [ "Catch-22" ],
        "_id" : "2"
      }
    } ], {
      "_index" : "library",
```

```

    "_type" : "book",
    "_id" : "3",
    "_score" : 194.6,
    "fields" : {
      "title" : [ "The Complete Sherlock Holmes" ],
      "_id" : "3"
    }
  }, {
    "_index" : "library",
    "_type" : "book",
    "_id" : "1",
    "_score" : 193.9,
    "fields" : {
      "title" : [ "All Quiet on the Western Front" ],
      "_id" : "1"
    }
  }, {
    "_index" : "library",
    "_type" : "book",
    "_id" : "6",
    "_score" : 191.4,
    "fields" : {
      "title" : [ "The Peasants" ],
      "_id" : "6"
    }
  }, {
    "_index" : "library",
    "_type" : "book",
    "_id" : "4",
    "_score" : 189.6,
    "fields" : {
      "title" : [ "Crime and Punishment" ],
      "_id" : "4"
    }
  }, {
    "_index" : "library",
    "_type" : "book",
    "_id" : "5",
    "_score" : 178.4,
    "fields" : {
      "title" : [ "The Sorrows of Young Werther" ],
      "_id" : "5"
    }
  }
]
}

```

此时读者不难发现，查询结果与预期的完全一致。

3. 额外说明

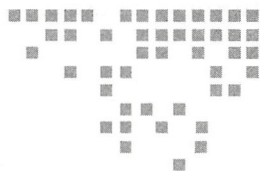
本节提供的范例比较简单，如果读者对 Lucene 表达式很感兴趣，可参考以下官方文

档：http://lucene.apache.org/core/4_9_0/expressions/index.html?org/apache/lucene/expressions/js/package-summary.html。该文档向读者展示 Lucene 的 expression 模块所提供的功能。

3.7 小结

在本章中，扩展了读者在查询处理与数据分析方面的知识。首先，介绍了查询二次评分，即如何对查询返回文档计算二次得分。同时也讨论了如何控制多匹配查询。然后介绍了两种重要的聚合类型：一种是提取返回结果中的重要词项（集），另一种是对文档进行分组（这是一种用户参与度极高的特性）。我们也探究了 Elasticsearch 中文档建模的不同方法。最后讨论了 Elasticsearch 中的脚本功能，以及自 1.0 版本以后的各种变化。

在下一章中，我们将探讨如何提升用户搜索体验。从拼写检查开始，介绍拼写检查的基本功能，以及如何将错误的查询纠正为正确的查询。之后将讨论在各种拼写错误的场景中使用合适的方法。最后，通过一个范例来介绍如何提升查询的相关性。我们将向读者展示一个返回较差结果的查询，并不断对其调优。



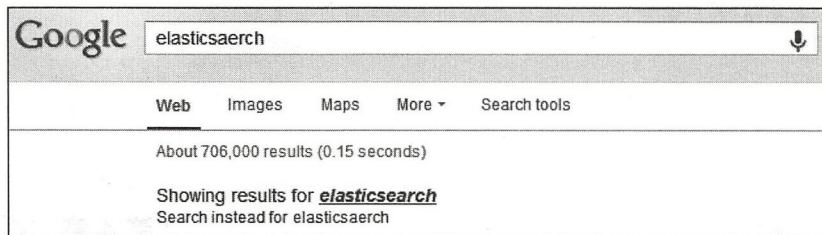
改善用户搜索体验

在上一章我们了解了查询处理及数据分析。首先，我们介绍了查询二次评分功能，它能重新计算查询返回文档的 top-N 文档的得分。然后学习了如何在 Elasticsearch 中控制多匹配，同时介绍了两种非常具有吸引力的聚合类型：significant_terms 聚合及 top_hits 聚合。还讨论了多种不同的文档关系建模技术。最后，我们了解了 Elasticsearch 的脚本模块，及 1.0 版本后脚本功能的变迁。本章，我们将聚焦在用户搜索体验上。到本章结束时，将涵盖以下内容：

- ❑ 如何使用 Elasticsearch Suggest API 改正用户的拼写错误
- ❑ 如何使用 term suggester 给出单词建议
- ❑ 如何使用 phrase suggester 提示完整词组
- ❑ 如何配置建议功能以匹配你的需求
- ❑ 如何使用 complete suggester 的自动补全功能
- ❑ 如何使用 Elasticsearch 的各种功能改进搜索相关性

4.1 改正用户拼写错误

改善用户搜索体验最简单的方式之一是纠正他们的拼写错误。要么自动地，要么仅显示正确的查询短语，并允许用户使用它。例如，当我们输入 elasticsaerch（正确拼写应是 Elasticsearch）时，Google 会这样提示我们：



自 0.90.0.Beta1 版本起, Elasticsearch 允许我们使用 Suggest API 改正用户拼写错误。不过, 相关文档指出这个功能仍在开发中。在即将问世的 Elasticsearch 新版本中, 它可能会发生巨大变化, 引入很多新特性。在本小节中我们将对 Elasticsearch 提供的 Suggest API 做完整的介绍, 同时会提供一些范例, 其中既有简单的案例, 也有需要较多配置的案例。

4.1.1 测试数据

为了阐述本节内容, 我们需要多准备一些文档。为了获取需要的数据, 我们决定使用 Wikipedia river 插件来索引一些 Wikipedia 上公开的文档。首先我们需要运行如下命令来安装这个插件:

```
bin/plugin -install elasticsearch/elasticsearch-river-wikipedia/2.4.1
```

接着执行如下命令:

```
curl -XPUT 'localhost:9200/_river/wikipedia_river/_meta' -d '{
  "type" : "wikipedia",
  "index" : {
    "index" : "wikipedia"
  }
}'
```

在此之后, Elasticsearch 开始从 Wikipedia 上下载英文文档并索引它们。

如果读者查看系统日志, 将会看到如下信息:

```
[2014-08-28 22:35:01,566] [INFO ] [river.wikipedia      ] [Thing]
[wikipedia] [Wikipedia_river] creating wikipedia stream river for
[http://download.wikimedia.org/enwiki/latest/enwiki-latest-pages-
articles.xml.bz2]
[2014-08-28 22:35:01,568] [INFO ] [river.wikipedia      ] [Thing]
[wikipedia] [Wikipedia_river] starting wikipedia stream
```

如读者所见, 该插件已开始工作。经过一些时间, 创建的 wikipedia 索引中已经有数据了。如果您想索引 Wikipedia 上所有最近的英文文档, 则需要点耐性, 这个比较耗时。出于演示的目的, 我们仅索引了部分文档, 当索引文档数到达 7080049 时, 我们停止了索引操作。此时索引大小为 19GB (没有使用副本)。

4.1.2 深入技术细节

Suggest API 从版本 0.90.3 开始被引入，但是它并不是 Elasticsearch 中最简单的 API。为了获得期望的建议信息，我们可以在查询中增加一个 suggest 节点，或者使用一个 Elasticsearch 提供的特殊的 REST 端点。此外，我们还拥有多个不同的 suggest 实现，用来纠正用户的拼写错误及创建自动补全等功能。以上功能给予我们一个强力且灵活的机制，用来使我们的搜索体验更佳。

当然，建议功能的效果跟我们的数据有关。如果索引中的文档数较少，可能就找不到合适的建议结果。当数据量较小时，Elasticsearch 索引中含有的词汇相对较少，因此能给出的候选建议结果也偏少。相反，数据量越大，我们拥有错误数据的可能性就越大。尽管如此，Elasticsearch 都能很好地处理这些情况。



注意 本章的布局结构与其他章节稍有不同。我们以一个简单例子作为本章开始。这个例子着重于告诉我们如何获取建议结果，以及如何解释 Suggest API 的响应，而不关注完整的配置选项。这是因为我们不想让你沉入过多的技术细节，而是想告诉你能从中得到什么。更多的配置参数稍后再谈。

4.1.3 suggester

在我们继续进行查询和分析响应结果之前，先简单交代一下可用的 suggester 类型。Elasticsearch 目前允许我们使用 3 种 suggester：一种是 term suggester，一种是 phrase suggester，还有一种是 complete（自动完成）suggester。前两种 suggester 可以用来改正拼写错误，而第 3 种 suggester 能够用来开发出迅捷且自动化的补全功能。不过目前，我们暂不聚焦于特定的 suggester。我们先看看查询的可能性和 Elasticsearch 的响应。我们将试着展示普遍原则，然后再深入探讨各种 suggester 的细节。

1. 使用 _suggest REST 终端

为获取给定文本的提示结果，第一个可行办法是使用精心设计的 _suggest REST 端点。我们需要提供分析的文本和 suggester 类型（term 或 phrase）。假如我们想得到关于 graphics designer（我们故意使用错误的拼写）的提示建议，需要执行如下查询：

```
curl -XPOST 'localhost:9200/wikipedia/_suggest?pretty' -d '{
  "first_suggestion" : {
    "text" : "wordl war ii",
    "term" : {
      "field" : "_all"
    }
  }
}
```


如你所见，每个 suggestion 请求都以对象（JSON 对象）的形式发送给 Elasticsearch。对象中包含我们指定的名字（在上面的例子中，名字是 first_suggestion）。我们用 text 参数指定想要查询的文本信息。最后，我们添加 suggester 对象，可以为 term 或 phrase 类型的 suggester。这些 suggester 对象拥有自己的配置。例如在之前的例子中，我们使用了 term suggester 类型，并通过 field 属性来指定建议从哪个字段里产生。

我们可以在一次请求中包含多个 suggestion 对象。每个对象拥有不同的名字。例如，如果我们要在上面的请求中增加一个针对单词 “raceing” 的 suggestion 对象，可以使用如下命令：

```
curl -XPOST 'localhost:9200/wikipedia/_suggest?pretty' -d '{
  "first_suggestion" : {
    "text" : "wordl war ii",
    "term" : {
      "field" : "_all"
    }
  },
  "second_suggestion" : {
    "text" : "raceing",
    "term" : {
      "field" : "text"
    }
  }
}'
```

2. 理解 _suggest REST 端点的响应

首先我们看看 _suggest REST 终端的响应示例。不同类型的 suggester 响应格式有所差异。仍然以之前使用的第一个命令为例，该命令中使用了词项类型 suggester，我们看看它的响应：

```
{
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "first_suggestion" : [ {
    "text" : "wordl",
    "offset" : 0,
    "length" : 5,
    "options" : [ {
      "text" : "world",
      "score" : 0.8,
      "freq" : 130828
    } ], {
      "text" : "words",
```

```

    "score" : 0.8,
    "freq" : 20854
  }, {
    "text" : "wordy",
    "score" : 0.8,
    "freq" : 210
  }, {
    "text" : "woudl",
    "score" : 0.8,
    "freq" : 29
  }, {
    "text" : "worde",
    "score" : 0.8,
    "freq" : 20
  } ]
}, {
  "text" : "war",
  "offset" : 6,
  "length" : 3,
  "options" : [ ]
}, {
  "text" : "ii",
  "offset" : 10,
  "length" : 2,
  "options" : [ ]
} ]
}

```

在响应的“first_suggestion”对象中，term suggerter 为每个 text 字段中的词返回一个建议列表。列表中包含可能的建议词以及一些附加信息。例如，从“wordl”这个词的响应数据中我们可以看到以下信息：请求原始词（text 参数），原始词在请求 text 中的偏移量（offset 参数）及长度（length 参数）。

options 数组包含给定词的建议词。如果 Elasticsearch 没有找到任何建议词，则 options 数组为空。该数组的每一项都包含一个建议词和以下可以用来表征该建议的信息：

- text: Elasticsearch 给出的建议词。
- score: 建议词的得分，得分越高的建议词其质量越高。
- freq: 建议词的文档频率。这里的频率指建议词在被查询索引的多少个文档中出现过。文档频率越高，说明包含这个建议词的文档也越多，并且这个词符合我们查询意图的可能性也越大。



注意 phrase suggerter 的响应结构和这里 term suggerter 的响应结构有所不同。我们将在本节的后面探讨 phrase suggerter 的响应。

3. 在查询请求中包含建议请求

除了使用 `_suggest` REST 端点，我们也可以在普通的 Elasticsearch 查询请求中包含建议请求。例如，我们可以通过如下方式，在查询请求中融入我们之前例子中的建议请求：

```
curl -XGET 'localhost:9200/wikipedia/_search?pretty' -d '{
  "query" : {
    "match_all" : {}
  },
  "suggest" : {
    "first_suggestion" : {
      "text" : "wordl war ii",
      "term" : {
        "field" : "_all"
      }
    }
  }
}'
```

如读者所期望的那样，上面这个查询的响应既包括了查询结果，也包括了建议结果，如下所示：

```
{
  "took" : 5,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 7080049,
    "max_score" : 1.0,
    "hits" : [
      ...
    ]
  },
  "suggest" : {
    "first_suggestion" : [ {
      "text" : "wordl",
      "offset" : 0,
      "length" : 5,
      "options" : [ {
        "text" : "world",
        "score" : 0.8,
        "freq" : 130828
      } ], {
        "text" : "words",
        "score" : 0.8,
```

```

      "freq" : 20854
    }, {
      "text" : "wordy",
      "score" : 0.8,
      "freq" : 210
    }, {
      "text" : "woudl",
      "score" : 0.8,
      "freq" : 29
    }, {
      "text" : "worde",
      "score" : 0.8,
      "freq" : 20
    } ]
  }, {
    "text" : "war",
    "offset" : 6,
    "length" : 3,
    "options" : [ ]
  }, {
    "text" : "ii",
    "offset" : 10,
    "length" : 2,
    "options" : [ ]
  } ]
}

```

上面的查询响应中包含了查询结果及查询建议，查询建议的消息体结构之前已经讨论过了。

还有一种可能场景：我们可能希望一次性获得针对同一段文本的多种类型的查询建议。这时候我们可以用 `suggest` 对象把建议请求封装起来，让 `text` 作为 `suggest` 对象的一个选项。例如：如果我们希望获取“wordl war ii”文本在 `text` 字段和 `_all` 字段中的建议，可以使用如下命令：

```

curl -XGET 'localhost:9200/wikipedia/_search?pretty' -d '{
  "query" : {
    "match_all" : {}
  },
  "suggest" : {
    "text" : "wordl war ii",
    "first_suggestion" : {
      "term" : {
        "field" : "_all"
      }
    },
    "second_suggestion" : {
      "term" : {

```


数的词项做一个区分对待，如果该词项不存在于索引中的，则返回它的建议词，否则不返回。如果本选项取值为 `popular`，则要求 Elasticsearch 在生成建议词时做一个判断，如果建议词比原词更受欢迎（在更多文档中出现），则返回，否则不返回。最后一个可用取值是 `always`，意思是每个 `text` 中的每个词生成建议词。

（3）term suggester 的其他配置选项

除了刚刚提到的通用配置选项，Elasticsearch 还提供一些仅适用于 term suggester 的选项。列举如下。

- ❑ `lowercase_terms`：如果本选项设置为 `true`，Elasticsearch 会把 `text` 文本分词得到的词项都转为小写。
- ❑ `max_edits`：默认值是 2，用来设定建议词与原始词的最大编辑距离。Elasticsearch 允许我们设置为 1 或 2。设置为 1 可能会得到较少的建议词，而对于有多个拼写错误的原始词，则可能没有建议词。一般来说，如果看到很多建议词项，可能是由于某些错误引起，此时可以将 `max_edits` 的值设置为 1。
- ❑ `prefix_len`：一般来说拼写错误不会出现在单词开头。Elasticsearch 允许我们设置建议词的开头几个字符必须和原始词开头字符匹配。这个选项的默认值为 1。如果我们正在与 suggester 的性能作战，可以通过增加这个取值来得到更好的性能，因为这样做会减少参与计算的建议词数量。
- ❑ `min_word_len`：这个选项用于指定可供返回的建议词的最少字符数。默认值是 4。
- ❑ `shard_size`：这个选项用于指定每个分片返回建议词的最大数量。默认等于 `size` 参数的值。如果给这个参数设定更大（大于 `size` 参数值）的值，会得到更精确的文档频率（因为词项分布在多个索引分片中，除非我们的索引只有一个分片），但是会导致拼写检查器的性能下降。
- ❑ `max_inspections`：这个选项用于控制 Elasticsearch 在一个分片中检查多少个候选者来产生可用的建议词，默认值是 5。Elasticsearch 针对每个原始词总共最多需要扫描 `shard_size * max_inspection` 个候选者。如果给这个选项设置更大（大于 5）的值，会提高精准度，但会降低性能。
- ❑ `min_doc_freq`：这个选项的默认值是 0，表示未启用。这个选项可以控制建议词的最低文档频率，只有文档频率高于本选项值的建议词才可以被返回（这个值是针对每个分片的，不是索引的全局取值）。例如，取值为 2 表示只有在给定分片中文档频率大于等于 2 的建议词才能被返回。把取值设置为大于 0 的数，可以提高返回提示词的质量，但是会让一些文档频率低于本值的建议词无法被输出。利用这个选项可以帮助我们去掉那些文档频率低、可能不正确的建议词。这个选项的取值也以设置为百分比，如果这样做，这个取值必须小于 1。例如，0.01 表示建议词的文档频率最低



不能小于当前分片文档数的 1% (当然, 该数值也是针对每个 shard 的)。

- ❑ `max_term_freq` : 这个选项用于设置 `text` 中词项的最大文档频率, 文档频率高于设定值的词项不会给出拼写纠错建议。默认值是 0.01。和 `min_doc_freq` 选项类似, 本选项的取值可以是精确数字 (例如 4 或者 100), 也可以是小于 1 的小数, 表示百分比 (例如, 0.01 表示 1%)。请记住, 这个值也是针对单个分片设定的。取值越高, 拼写检查器的性能越好。一般来说, 如果我们想要在拼写检查时排除掉高频词, 这个参数非常有用, 因为高频词往往不会存在拼写错误。
- ❑ `accuracy` : 这个选项取值范围是 0 ~ 1, 默认值是 0.5。这个选项指定建议词和原词的相似度。取值越高, 相似度越高。这个值用于在计算编辑距离时与原始词做比较。
- ❑ `string_distance` : 这个选项是个高级设置, 用于指定计算词项相似度的算法。支持以下算法: `internal`, `damerau_levenshtein`, `levenshtein`, `jarowinkler` 以及 `ngram`。`internal` 比较算法基于 Damerau Levenshtein 相似度算法的优化实现。`damerau_levenshtein` 是 Damerau Levenshtein 字符串距离算法 (http://en.wikipedia.org/wiki/Damerau-Levenshtein_distance) 的实现。`levenshtein` 是 Levenshtein 距离算法 (http://en.wikipedia.org/wiki/Levenshtein_distance) 的实现。`jarowinkler` 是 Jaro-Winkler 距离算法 (http://en.wikipedia.org/wiki/Jaro-Winkler_distance) 的实现。`ngram` 是基于 n-gram 距离的算法实现。



注意 因为我们在之前已经用 `term suggester` 做过示例, 这里我们就不再演示如何使用 `term suggester` 以及其响应信息的格式。你可以回到本节开头阅读这些信息。

5. phrase suggester

`term suggester` 提供了一种基于单个词项的拼写纠错方法。然而, 当我们想要得到短语建议时, 它就不能胜任了。所以我们引入了 `phrase suggester`。`phrase suggester` 建立在 `term suggester` 之上, 并添加了额外的短语计算逻辑, 因此可以返回完整的短语建议而不是单个词项的建议。它基于 n-gram 语言模型计算建议项的质量, 在短语纠错方面它是比 `term suggester` 更好的选择。n-gram 方法将索引中的词项切分成 gram。gram 指由一个或多个字母组成的单词片段。例如, 我们将 `mastering` 切分成 bi-grams (两个字母的 n-gram), 切分结果如下: `ma as st te er ri in ng`。



注意 你可以阅读后面这篇维基百科的文章来了解更多关于 n-gram 模型的知识: http://en.wikipedia.org/wiki/Language_model#N-gram_models。



(1) 使用示例

在展示所有可能性之前，我们需要先配置一下 phrase suggester。首先我们演示一下如何使用它。我们可以执行如下命令，向 `_search` 端点发送一个仅含有 suggests 片段的简单查询请求：

```
curl -XGET 'localhost:9200/wikipedia/_search?pretty' -d '{
  "suggest" : {
    "text" : "wordl war ii",
    "our_suggestion" : {
      "phrase" : {
        "field" : "_all"
      }
    }
  }
}'
```

这段代码几乎和使用 term suggester 查询时的代码一模一样，只是用 phrase 类型替代了 term 类型。这段代码的响应信息如下：

```
{
  "took" : 58,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 7080049,
    "max_score" : 1.0,
    "hits" : [
      ...
    ]
  },
  "suggest" : {
    "our_suggestion" : [ {
      "text" : "wordl war ii",
      "offset" : 0,
      "length" : 12,
      "options" : [ {
        "text" : "world war ii",
        "score" : 7.055394E-5
      }, {
        "text" : "words war ii",
        "score" : 2.3738032E-5
      }, {
        "text" : "wordy war ii",
        "score" : 3.575829E-6
      }, {
```




```

      "text" : "worde war ii",
      "score" : 1.1586584E-6
    }, {
      "text" : "woudl war ii",
      "score" : 1.0753317E-6
    } ]
  } ]
}

```

我们看到，响应信息也和 term suggester 的响应信息非常相似。不过这里返回的是完整的短语建议，而不是针对单个词项的建议。建议项列表默认按得分排序。我们同样可以在 phrase 片段中配置附加参数。接下来我们就看看都有那些可用的配置选项。

(2) 配置

phrase suggester 的配置项分为三组：基本参数，用来定义一般表现；平滑参数，用来平衡 n-gram 权重；候选者生成器参数，负责生成各个词项的建议列表，这些列表被用来生成最终的短语建议。



注意 因为 phrase suggester 是建立在 term suggester 之上的，所以它可以使用 term suggester 的一些配置选项，包括：text、size、analyzer 和 shard_size。请参考本章之前关于 term suggester 的描述来了解这些参数的含义。

(3) 基本配置

除了前面提到的这几个选项之外，phrase suggester 对外提供如下基本配置项。

- ❑ **highlight**：该选项可设置建议项高亮处理。该选项需要结合 pre_tag 及 post_tag 属性使用，这两个属性是可配置的，返回项将会被 pre_tag 和 post_tag 括起来。例如，这两者可以被设置为 和 ，返回项将会被显示为高亮。
- ❑ **gram_size**：这个选项指定与 field 参数对应字段中存储的 n-gram 的最大的 n。如果指定字段中没存储 n-gram，这个值应该被设置为 1，或者根本不用在请求中携带这个参数。如果这个值没有设置，Elasticsearch 会尝试自己去探测出正确的值。例如，对于使用 shingle 过滤器 (<http://www.Elasticsearch.org/guide/reference/index-modules/analysis/shingle-tokenfilter/>) 的字段，这个值会被设置为 max_shingle_size 属性的取值（如果没有显式设置）。
- ❑ **confidence**：使用这个选项可以基于得分来限制返回的建议项。选项值被作用到输入短语的原始得分上（原始得分乘以这个值），得到新的得分。新的得分作为临界值用于限制生成的建议项。如果建议项的得分高于这个临界值，它可以被放入输出结果列表，否则被丢弃。例如，取值 1.0（本选项默认值）意味着只有得分高于输入短语



的建议项才会被输出。另一方面，设置为 0.0 表示输出所有建议项（个数受 size 参数的限制），而不管它们的得分高低。

- ❑ **max_errors**：这个属性用于指定拼写错误词项的最大个数或百分比。取值可以是一个整数，例如 1、5，或者一个 0 ~ 1 的浮点数。浮点数会被解释成百分比，表示最多可以有百分之多少的词项含有拼写错误。例如，0.5 代表 50%。而如果取值为整数，比如 1、5，Elasticsearch 会把它当作拼写错误词项的最大个数。默认值是 1，意思是最多只能有一个词项含有拼写错误。
- ❑ **separator**：这个选项用于指定 bigram 字段中词项间的分隔符。默认分隔符是空格。
- ❑ **force_unigrams**：这个选项用于指定拼写检查器是否强制使用一元语法模型（unigram）。默认值为 true。
- ❑ **token_limit**：这个选项用于指定建议列表最多可包含的词项数。默认值是 10。设置为更高的值能够提升建议精准度，不过需要付出性能下降的代价。
- ❑ **collate**：该选项允许用户检查特定查询（在 collate 对象内部使用 query 属性）或过滤器（在 collate 对象内部使用 filter 属性）返回建议项的每一项。这里的查询或过滤实际上是一个模板，对外暴露一个 {{suggestion}} 变量，该变量代表当前正在处理的建议。在 collate 对象中添加 prune 属性，将其值设置为 true，Elasticsearch 将会将建议项与查询或过滤器的匹配信息包含进来（这些信息被包含在返回结果的 collate_match 属性中）。除此之外，如果使用了 preference 属性，查询偏好信息也会被包含进返回结果中（可使用普通查询中的同名参数的值）。
- ❑ **real_word_error_likelihood**：这个选项用于设定词项有多大可能拼写错误，尽管它存在于索引的词典中。选项取值是百分比，默认值为 0.95，用于告知 Elasticsearch 它的词典中约有 5% 的词项拼写不正确。减小这个值意味着更多的词项会被认为含有拼写错误，尽管它们可能是正确的。

现在，我们来查看一个范例，该范例中用到了上面所提及的这些参数。我们修改一下之前的那个 phrase suggestion 查询，添加粗体显示。命令如下：

```
curl -XGET 'localhost:9200/wikipedia/_search?pretty' -d '{
  "suggest" : {
    "text" : "wordl war ii",
    "our_suggestion" : {
      "phrase" : {
        "field" : "_all",
        "highlight" : {
          "pre_tag" : "<b>",
          "post_tag" : "</b>"
        }
      }
    }
  }
}
```



```

    "collate" : {
      "prune" : true,
      "query" : {
        "match" : {
          "title" : "{{suggestion}}"
        }
      }
    }
  }
}
}
}'

```

上面这个查询的返回结果如下所示：

```

{
  "took" : 3,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 7080049,
    "max_score" : 1.0,
    "hits" : [
      ...
    ]
  },
  "suggest" : {
    "our_suggestion" : [ {
      "text" : "wordl war ii",
      "offset" : 0,
      "length" : 12,
      "options" : [ {
        "text" : "world war ii",
        "highlighted" : "<b>world</b> war ii",
        "score" : 7.055394E-5,
        "collate_match" : true
      }, {
        "text" : "words war ii",
        "highlighted" : "<b>words</b> war ii",
        "score" : 2.3738032E-5,
        "collate_match" : true
      }, {
        "text" : "wordy war ii",
        "highlighted" : "<b>wordy</b> war ii",
        "score" : 3.575829E-6,
        "collate_match" : true
      }
    ]
  }
}

```



```

    }, {
      "text" : "worde war ii",
      "highlighted" : "<b>worde</b> war ii",
      "score" : 1.1586584E-6,
      "collate_match" : true
    }, {
      "text" : "woudl war ii",
      "highlighted" : "<b>woudl</b> war ii",
      "score" : 1.0753317E-6,
      "collate_match" : true
    } ]
  } ]
}

```

(4) 配置平滑模型

平滑模型 (smoothing model) 是 phrase suggester 的一个功能。它的职责是平衡索引中不存在的稀有 n-gram 词元和索引中存在的高频 n-gram 词元之间的权重。这是个非常高级的选项。如果你修改它, 你应该检查一下查询建议的响应信息, 看看它是不是满足你的需求。平滑技术被用于语言模型中, 用来避免某些词项的出现零概率的情况。Elasticsearch 的 phrase suggester 支持多种平滑模型。



注意 通过以下链接你可以了解更多语言模型的信息: http://en.wikipedia.org/wiki/Language_model。

为了选择使用某个平滑模型, 我们需要在请求中添加一个 smoothing 对象, 并让它包含一个我们要使用的平滑模型名称。当然我们也可以根据需要设置平滑模型的各种属性。例如, 我们可以执行如下命令:

```

curl -XGET 'localhost:9200/wikipedia/_search?pretty&size=0' -d '{
  "suggest" : {
    "text" : "wordl war ii",
    "generators_example_suggestion" : {
      "phrase" : {
        "analyzer" : "standard",
        "field" : "_all",
        "smoothing" : {
          "linear" : {
            "trigram_lambda" : 0.1,
            "bigram_lambda" : 0.6,
            "unigram_lambda" : 0.3
          }
        }
      }
    }
  }
}

```




```

    }
  }
}
}'

```

Elasticsearch 共提供 3 种可用的平滑模型。让我们看看 Elasticsearch 中可供 phrase suggester 使用的平滑模型。

Stupid backoff 是 Elasticsearch 的 phrase suggester 默认的平滑模型。为了能够修改它或强制使用它，我们需要在请求中使用它的名称 `stupid_backoff`。Stupid backoff 平滑模型的实现是这样的：如果高阶的 n -gram 出现频率为 0，它会转而使用低阶的 n -gram 的频率（并且给该频率打个折扣，折扣率由 `discount` 参数指定）。举例来说，假定有一个 bigram `ab` 和一个 unigram `c`。`ab` 和 `c` 普遍存在于我们的索引中，而索引中不存在 trigram `abc`。这种情况下 Stupid backoff 模型会直接使用 `ab` 二元分词模型，并且给它一个与 `discount` 属性值相同的折扣。

Stupid backoff 模型只提供了一个 `discount` 参数供我们调整。`discount` 参数的默认值是 0.4，被用来给低阶的 n -gram 打折。

你可以访问以下网址来获取更多关于 N 元语法模型的信息：http://en.wikipedia.org/wiki/N-gram#Smoothing_techniques 以及 http://en.wikipedia.org/wiki/Katz's_back-off_model（该模型和 stupid backoff 模型类似）。

Laplace 平滑又称为加法平滑（additive smoothing）。当我们使用它时（为使用该模型，我们需要使用 `laplace` 作为模型的名字），由 `alpha` 参数指定的常量（默认值 0.5）将被加到词项的频率上，用来平衡频繁和不频繁的 n -gram。之前提到过，Laplace 平滑模型可通过 `alpha` 参数进行配置。Alpha 参数默认值为 0.5，取值通常等于或小于 1.0。

可以在 http://en.wikipedia.org/wiki/Additive_smoothing 这个网页中了解更多关于加法平滑的信息。

线性插值是这里介绍的最后一种平滑模型。它使用配置中提供的 `lambda` 值计算 trigram、bigram 及 unigram 的权重。为了使用线性插值平滑模型，我们需要在查询对象中指定 `smoothing` 为 `linear`，并提供 3 个参数：`trigram_lambda`、`bigram_lambda` 和 `unigram_lambda`。以上 3 个参数之和必须为 1。每个参数对应一种 N 元分词类型。比如，`bigram_lambda` 将被用作 bigram 的权重。

（5）配置候选生成器

为了给 `text` 参数文本中的每个 term 返回可能的建议项，Elasticsearch 使用被称为候选生成器的工具。你可以把候选生成器当作 term suggester 来理解，不过实际上它们不是一回事。它们很相似，因为它们都被用在每个单独 term 上。返回的候选 term 将和查询文本中其



他 term 的建议词的得分合并，通过这种方式最终生成短语建议。

直接生成器 (direct generator) 是目前 Elasticsearch 中唯一可用的候选生成器，尽管在未来可能会有更多其他的候选生成器加入进来。Elasticsearch 允许在一个短语建议请求中指定多个直接生成器。我们可以通过设置名为 `direct_generators` 的列表来做到这一点。例如，我们可以执行如下命令：

```
curl -XGET 'localhost:9200/wikipedia/_search?pretty&size=0' -d '{
  "suggest" : {
    "text" : "wordl war ii",
    "generators_example_suggestion" : {
      "phrase" : {
        "analyzer" : "standard",
        "field" : "_all",
        "direct_generator" : [
          {
            "field" : "_all",
            "suggest_mode" : "always",
            "min_word_len" : 2
          },
          {
            "field" : "_all",
            "suggest_mode" : "always",
            "min_word_len" : 3
          }
        ]
      }
    }
  }
}
```

响应信息和之前非常相似，我们在此不做展示。

(6) 配置直接生成器

用于配置直接生成器行为的参数和 term suggester 暴露的参数相似。它们共同参数有：field (必选参数)、size、suggest_mode、max_edits、prefix_length、min_word_len (这里默认为 4)、max_inspections、min_doc_freq、max_term_freq。请查阅 term suggester 相关内容来了解这些参数的含义。

除了以上提及的配置属性，直接生成器还支持 `pre_filter` 和 `post_filter` 参数。这两个参数可以用来向 Elasticsearch 提供一个分析器的名称。Pre_filter 参数指定的分析器用于处理传入直接生成器的词项，而 post_filter 参数指定的分析器用于处理由直接生成器输出的词项。处理操作在词项被传递给短语评分器 (phrase scorer) 之前进行。



我们可以使用直接生成器的过滤功能（通过设置 `pre_filter`）来在建议项被传递给直接生成器之前添加一些它们的同义词。例如，让我们更新一下 `wikipedia` 索引的设置以支持简单同义词，然后把这些同义词用在过滤功能中。使用下面这些命令：

```
curl -XPOST 'localhost:9200/wikipedia/_close'
curl -XPUT 'localhost:9200/wikipedia/_settings' -d '{
  "settings" : {
    "index" : {
      "analysis": {
        "analyzer" : {
          "sample_synonyms_analyzer": {
            "tokenizer": "standard",
            "filter": [
              "sample_synonyms"
            ]
          }
        },
        "filter": {
          "sample_synonyms": {
            "type": "synonym",
            "synonyms" : [
              "war => conflict"
            ]
          }
        }
      }
    }
  }
}
```

```
curl -XPOST 'localhost:9200/wikipedia/_open'
```

我们需要先关闭索引，然后更新索引设置，再重新打开它，因为 Elasticsearch 不允许修改已打开索引的配置。现在可以测试直接生成器的同义词功能了。使用如下命令：

```
curl -XGET 'localhost:9200/wikipedia/_search?pretty&size=0' -d '{
  "suggest" : {
    "text" : "wordl war ii",
    "generators_with_synonyms" : {
      "phrase" : {
        "analyzer" : "standard",
        "field" : "_all",
        "direct_generator" : [
          {
```



```

    "field" : "_all",
    "suggest_mode" : "always",
    "post_filter" : "sample_synonyms_analyzer"
  }
]
}
}
}'

```

该命令响应结果如下:

```

{
  "took" : 47,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 7080049,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "suggest" : {
    "generators_with_synonyms" : [ {
      "text" : "wordl war ii",
      "offset" : 0,
      "length" : 12,
      "options" : [ {
        "text" : "world war ii",
        "score" : 7.055394E-5
      }, {
        "text" : "words war ii",
        "score" : 2.4085322E-5
      }, {
        "text" : "world conflicts ii",
        "score" : 1.4253577E-5
      }, {
        "text" : "words conflicts ii",
        "score" : 4.8214292E-6
      }, {
        "text" : "wordy war ii",
        "score" : 4.1216194E-6
      } ]
    } ]
  } ]
}

```

可以看出, 这里 phrase suggester 返回的不是 war 词项, 而是 conflict 词项。这正是我



们在本例中想要的效果。我们的同义词配置生效了。不过，请记住，同义词扩展发生在对建议项打分之前，所以可能出现这种情况：同义词的建议项打分并不一定是最高的，你可能无法在建议结果中看到它们。

6. completion suggester

Elasticsearch 0.90.3 版本的发布给予我们使用一个特殊 suggester 的机会，该 suggester 基于前缀匹配。使用该 suggester 我们可以很容易开发出自动完成功能，因为复杂的数据结构都存储在索引中，不用在查询时实时计算。尽管这个高效的 suggester 不是关于拼写纠错的，我们还是认为，用一个简单示例来介绍一下它将会对你有所帮助。

(1) completion suggester 背后的逻辑

基于前缀的 suggester 构建在一种称为 FST (Finite State Transducer)(http://en.wikipedia.org/wiki/Finite_state_transducer) 的数据结构之上。它十分高效，但是构建它的资源消耗却非常显著，特别是在拥有大量数据的时候。如果我们在某些节点上构建这些数据结构，每当节点重启或集群状态变更时，都会付出性能代价。鉴于这个问题，Elasticsearch 的设计者们决定在索引过程中创建类似 FST 的数据结构，并把它存储在索引中，在需要的时候可以把它加载进内存。

(2) 使用 completion suggester

为了使用基于前缀的 suggester，我们需要使用 completion 类型的字段来索引数据。这种类型的字段可以在索引中存储类似 FST 的数据结构。为了展示这个 suggester 的使用，假设我们要添加一个针对书籍作者的自动完成功能。除了作者名称外，我们还想返回该作者所写的书的 ID，我们通过一个额外查询来查找这些数据。首先使用如下命令建立 authors 索引：

```
curl -XPOST 'localhost:9200/authors' -d '{
  "mappings" : {
    "author" : {
      "properties" : {
        "name" : { "type" : "string" },
        "ac" : {
          "type" : "completion",
          "index_analyzer" : "simple",
          "search_analyzer" : "simple",
          "payloads" : true
        }
      }
    }
  }
}
```

该索引包括一个名为 `author` 的类型。每个文档有两个字段：`name` 和 `ac`。`name` 字段存储作者名字，`ac` 字段是我们用来实现自动完成的字段。这里我们关心的是 `ac` 字段。我们定义它为 `complete` 类型，该类型表示在索引中存储类似 FST 的数据结构。我们在索引和查询时都使用 `simple` 分析器。最后需要提及的是 `payload` 附加信息，它将伴随查询建议一起输出。本例中 `payload` 是书籍 ID 的数组。



注意 将要使用自动完成功能的字段，其类型是强制提供的，必须是 `completion`。默认情况下，`search_analyzer` 和 `index_analyzer` 设置为 `simple`，而 `payload` 属性设置为 `false`。

(3) 索引数据

为了索引数据，我们需要提供一些额外信息。让我们看看下面这个命令，该命令将索引两个描述作者信息的文档：

```
curl -XPOST 'localhost:9200/authors/author/1' -d '{
  "name" : "Fyodor Dostoevsky",
  "ac" : {
    "input" : [ "fyodor", "dostoevsky" ],
    "output" : "Fyodor Dostoevsky",
    "payload" : { "books" : [ "123456", "123457" ] }
  }
}'

curl -XPOST 'localhost:9200/authors/author/2' -d '{
  "name" : "Joseph Conrad",
  "ac" : {
    "input" : [ "joseph", "conrad" ],
    "output" : "Joseph Conrad",
    "payload" : { "books" : [ "121211" ] }
  }
}'
```

注意一下 `ac` 字段的数据结构。我们提供了 `input`、`output` 和 `payload` 属性。`payload` 属性用于提供查询返回的额外信息。`input` 属性提供的数据用于构建类 FST 数据结构，还用来匹配用户输入，以决定当前文档是否需要被返回。`output` 属性用于告知 `suggester` 返回文档中应包含什么数据。



注意 请记住，`payload` 必须是一个 JSON 对象，以 “{” 符号开头并以 “}” 符号结尾。

如果你的 `input` 和 `output` 属性取值相同，且不需要存储 `payload`，那么你可以如平常一样索引数据。例如：可使用如下命令索引前面例子中的第一个文档：

```
curl -XPOST 'localhost:9200/authors/author/3' -d '{
  "name" : "Stanislaw Lem",
  "ac" : [ "Stanislaw Lem" ]
}'
```

(4) 查询数据

最后我们看看如何查询刚刚索引的数据。假如我们想要找到作者名以 fyo 开头的文档，可以使用如下命令：

```
curl -XGET 'localhost:9200/authors/_suggest?pretty' -d '{
  "authorsAutocomplete" : {
    "text" : "fyo",
    "completion" : {
      "field" : "ac"
    }
  }
}'
```

在查看结果之前，我们先探讨一下查询本身。可以看出，我们发送请求的目标是 `_suggest` 端点，因为我们在这里不想发送一个标准查询，而仅仅对自动完成结果感兴趣。查询的其他部分和标准的建议查询如出一辙。查询类别需要设置为 `completion`。

之前命令的执行结果如下：

```
{
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "authorsAutocomplete" : [ {
    "text" : "fyo",
    "offset" : 0,
    "length" : 3,
    "options" : [ {
      "text" : "Fyodor Dostoevsky",
      "score" : 1.0,
      "payload":{"books":["123456","123457"]}
    } ]
  } ]
}
```

可以看出，我们从响应中获得了想要的文档。文档中包括 `payload` 信息，`payload` 包含了某作者的图书的 ID（列表）。

(5) 自定义权重

默认情况下，词项频率（term frequency）将被基于前缀的 `suggester` 作为文档权重。然

而，当你拥有多个索引分片或者你的索引由多个索引段组成时，这可能不是最好的方案。在这些情况下，自定义权重是有价值的。一般通过给 completion 类型的字段指定 weight 属性来实现。weight 属性取值应该设置为整数，而不是浮点数，与查询 boost、文档 boost 的情况类似。weight 取值越大，建议项的重要性越大。这项功能给予我们很多调整建议项排序的机会。

例如，假定我们想要给前面例子中的第一个文档指定权重。使用如下命令：

```
curl -XPOST 'localhost:9200/authors/author/1' -d '{
  "name" : "Fyodor Dostoevsky",
  "ac" : {
    "input" : [ "fyodor", "dostoevsky" ],
    "output" : "Fyodor Dostoevsky",
    "payload" : { "books" : [ "123456", "123457" ] },
    "weight" : 80
  }
}'
```

然后，如果执行刚才的查询，结果将是：

```
{
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "authorsAutocomplete" : [ {
    "text" : "fyo",
    "offset" : 0,
    "length" : 3,
    "options" : [ {
      "text" : "Fyodor Dostoevsky",
      "score" : 80.0,
      "payload": {"books": ["123456", "123457"]}
    } ]
  } ]
}
```

查看一下返回结果得分的变化。在最初的例子中，得分是 1.0，而现在得分是 80.0。因为我们在索引时设定 weight 参数为 80。

（6）额外参数

基于前缀的 suggester 还有 3 个额外参数我们尚未提及：max_input_length、preserve_separators 和 preserve_position_increments。后两个属性都可以设置为 true 或 false。如果把 preserve_separators 设置为 false，suggester 将忽略如空格之类的分隔符（当然，需要合适的分析器）。而如果建议项的第一个单词是停用词并且我们使用了过滤停用词的分析器，则

需要把 `preserve_position_increments` 设置为 `false`。例如，文档内容为 “The Clue”，这时 “The” 将被分析器丢弃。通过设置 `preserve_position_increments` 为 `false`，`suggester` 将通过查询 “c” 来返回这个文档。

而 `max_input_length` 参数默认设置为 50，该属性确定了能输入的最大的 UTF-16 字符串长度。该属性用于索引期限制内部结构能存储的最大字符数。

4.2 改善查询相关性

事实上，包括 Elasticsearch 在内的搜索引擎通常都是用来提供搜索服务的。某些特定情况下，只需要查看索引的一部分数据，更一般的情况下我们需要使用查询相关的所有数据，因而引入了评分机制。我们在 2.1 节中也提到过这一点。Elasticsearch 利用了 Apache Lucene 本身的评分功能，并允许我们使用多种查询类型来控制查询结果的得分。我们甚至可以修改底层评分算法，这一点我们将在第 6 章中提到。

有了这些功能之后，在设计查询时，我们通常可以找到最简单的查询来满足查询需求。尽管我们可以在 Elasticsearch 中做很多事，但一旦涉及评分控制，这些查询的结果可能就不是最有利于提升用户搜索体验了。这是因为 Elasticsearch 猜不出我们的业务逻辑是什么，也不知道从我们使用者的角度看，哪些文档是最好的。本节我们将追踪一个实际的查询相关性调优的例子。我们想让本章内容稍微有些与众不同，不是简单地把知识点告诉你，而是提供一个关于查询调优过程运作的完整示例。尽管本节某些例子是通用的，在你把它们移植到自己的应用中时，请确保它们有实际意义。

这里给一点小小的剧透，我们首先将执行一个简单的查询并返回我们想要的结果，然后修改这个查询，引入不同的 Elasticsearch 查询来使结果更好，我们会使用过滤器，还会降低垃圾文档的得分，之后我们将引入切面计算，用来提供下拉菜单让用户缩小查询结果范围。

4.2.1 数据

当然，为了展示查询修改后的返回结果，我们需要数据。我们乐于分享现实工作中的真实数据，不过我们不能这么做，原因可想而知。不过还有另一个解决办法：为了本节的演示目的，我们决定索引 Wikipedia 的数据。为了做到这一点，我们需要重复使用与 4.1 节中相同的那个 Wikipedia river。

如果不存在名为 `wikipedia` 的索引，`Wikipedia river` 将为我们创建。不过，该索引已经存在了，我们需要删掉它。尽管使用了相同的索引，但是索引字段需要做一些调整，因为我们需要添加一些额外的分析逻辑，另外为了重新索引数据，需要先创建索引。



注意

添加新 river 前需要移除旧 river。为了移除旧 river，需要执行下面这个命令：`curl -XDELETE 'localhost:9200/_river/wikipedia_river'`

为了重新导入文档，需执行下面这个命令：

```
curl -XDELETE 'localhost:9200/wikipedia'
curl -XPOST 'localhost:9200/wikipedia' -d'{
  "settings": {
    "index": {
      "analysis": {
        "analyzer": {
          "keyword_ngram": {
            "filter": [
              "lowercase"
            ],
            "tokenizer": "ngram"
          }
        }
      }
    }
  },
  "mappings": {
    "page": {
      "properties": {
        "category": {
          "type": "string",
          "fields": {
            "untouched": {
              "type": "string",
              "index": "not_analyzed"
            }
          }
        }
      }
    },
    "disambiguation": {
      "type": "boolean"
    },
    "link": {
      "type": "string",
      "index": "not_analyzed"
    },
    "redirect": {
      "type": "boolean"
    }
  }
}
```

```

    "redirect_page": {
      "type": "string"
    },
    "special": {
      "type": "boolean"
    },
    "stub": {
      "type": "boolean"
    },
    "text": {
      "type": "string"
    },
    "title": {
      "type": "string",
      "fields": {
        "ngram": {
          "type": "string",
          "analyzer": "keyword_ngram"
        },
        "simple": {
          "type": "string",
          "analyzer": "simple"
        }
      }
    }
  }
}
}'

```

现在我们在索引中创建了一个 page 类型的映射，用它来表示一个 Wikipedia 的页面。我们将在两个字段中执行查询：text 和 title。text 字段存储页面内容，而 title 字段存储页面标题。

接下来我们需要启动 Wikipedia river。我们想要获得最新的数据。使用如下命令来实例化 river 并索引数据：

```

curl -XPUT 'localhost:9200/_river/wikipedia/_meta' -d '{
  "type" : "wikipedia"
}'

```

以上就是我们要做的全部操作。Elasticsearch 将自动把最新的 Wikipedia 数据索引到名为 wikipedia 的索引中。我们只须耐心等待即可。不过，我们显然不够耐心：我们决定仅索

引前 1000 万文档。当 river 索引的页面数量达到这个值时，我们就停止 river。通过执行如下命令检查最终的文档数：

```
curl -XGET 'localhost:9200/wikipedia/_search?q=*&size=0&pretty'
```

响应结果如下：

```
{
  "took" : 5,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 10425136,
    "max_score" : 0.0,
    "hits" : [ ]
  }
}
```

可以看出，一共索引了 10 425 136 个文档。



注意 在运行本章中的示例代码时，请注意，我们用来建立索引的数据是随着时间不断变化的。因此本章这些示例的结果可能和书中描述的不一样。

4.2.2 改善相关性的探索之旅

准备好索引数据后，我们就可以搜索了。我们从头开始，先使用一个简单查询来获取感兴趣的结果，然后再尝试逐渐改善查询相关性。我们还将关注查询性能的改变，这些改变在我们优化相关性过程中很可能发生。

1. 标准查询

如你所知，Elasticsearch 默认把文档内容存入 `_all` 字段中。既然如此，我们为什么还要为如何同时查询多个字段伤脑筋呢？仅仅使用一个 `_all` 字段就可以了，对吗？按照这个思路，假定我们构建了如下查询，并使用它来获取感兴趣的数据：

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d'
{
  "query": {
    "match": {
      "_all": {
        "query": "australian system",
        "operator": "OR"
      }
    }
  }
}
```



```
    }
  }
}
```

因为我们只想获取 title 字段的内容（因为 title 字段设置为不存储，Elasticsearch 将使用 _source 字段来返回 title 字段的内容），我们在命令中添加了 fields=title 请求参数。当然，我们还希望结果是方便人类阅读的，因此还添加了 pretty 参数。

然而，结果并不是如我们期望的那样完美。返回结果第一页中的文档如下：

```
Australian Honours System
List of Australian Awards
Australian soccer league
Australian football league system
AANBUS
Australia Day Honours
Australian rating system
TAAATS
Australian Arbitration system
Western Australian Land Information System (WALIS)
```

来看一下文档标题。发现某些同时包含两个词项的文档被排在了后面。这个搜索结果多少有些差强人意。让我们来尝试改善一下。

2. 多匹配查询

首先我们要做的是，不再使用 _all 字段，因为我们想让 Elasticsearch 明白各字段的权重。例如，在这里 title 字段比存储页面内容的 text 字段更重要。为了让 Elasticsearch 明白这一点，我们需要使用 multi_match 查询。发送如下命令给 Elasticsearch：

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d'
{
  "query": {
    "multi_match": {
      "query": "australian system",
      "fields": [
        "title^100",
        "text^10",
        "_all"
      ]
    }
  }
}
```

返回结果第一页中的文档如下（完整的响应内容保存在随本书一起提供的 response_

query_multi_match.json 文件中):

```
Australian Antarctic Building System
Australian rating system
Australian Series System
Australian Arbitration system
Australian university system
Australian Integrated Forecast System
Australian Education System
The Australian electoral system
Australian preferential voting system
Australian Honours System
```

在这里我们舍弃了针对单个 `_all` 字段的查询, 转而选择同时查询 `title`、`text` 和 `_all` 字段。我们还做了加权处理: `boost` 值越大, 则该字段越重要 (字段 `boost` 的默认值为 1.0)。我们在此声明 `title` 字段比 `text` 字段重要, `text` 字段比 `_all` 字段重要。

再回头看查询结果, 它们看起来比之前的结果更相关了, 不过还是没有我们期望的那样好。例如, 比较一下前两个文档, 第 1 个文档的标题是 “Australian Antarctic Building System”, 而第 2 个文档的标题是 “Australian rating system”。我希望第 2 个文档的得分更高。

3. 引入短语查询

接下来我们应该想到的是引入短语查询。短语查询可以搞定刚才描述的问题。不过, 我们还是希望能够在与短语匹配的查询结果后面列出那些没有包含完整短语的结果。为此, 我们需要修改查询, 在最顶部加上一个布尔查询 (bool query)。当前的查询将放入 `must` 区域, 短语查询将放在 `should` 区域。修改后的查询示例如下:

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d'
{
  "query": {
    "bool": {
      "must": [
        {
          "multi_match": {
            "query": "australian system",
            "fields": [
              "title^100",
              "text^10",
              "_all"
            ]
          }
        }
      ]
    }
  },
  "should": [
```

```

    {
      "match_phrase": {
        "title": "australian system"
      }
    },
    {
      "match_phrase": {
        "text": "australian system"
      }
    }
  ]
}

```

查询结果的前几条如下：

```

Australian honours system
Australian Antarctic Building System
Australian rating system
Australian Series System
Australian Arbitration system
Australian university system
Australian Integrated Forecast System
Australian Education System
The Australian electoral system
Australian preferential voting system

```

尽管结果比之前的要好一点，不过与我们的期望还是有点距离。因为没有找到和所有短语匹配的记录。我们可以考虑引入 `slop` 参数。`slop` 参数用来设置单词间的最大间隔，在最大间隔之内的单词可以被认为与查询中的短语匹配。例如，我们这里使用的“australian system”短语查询，如果设置 `slop` 为大于等于 1 的数，可以和标题为“australian education system”的文档匹配上。让我们执行一条带 `slop` 参数的查询命令：

```

curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d'
{
  "query": {
    "bool": {
      "must": [
        {
          "multi_match": {
            "query": "australian system",
            "fields": [
              "title^100",
              "text^10",

```


殊文档（例如，那些被标记为已删除的文档）。我们在这里引入一个过滤器。过滤器的引入不会干扰其他文档的排序（因为过滤器不具备评分功能），反而可以缓存被过滤的结果（这一点由 Elasticsearch 自动完成），并使这些结果能够被之后的查询再次使用。带过滤器的查询命令如下：

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d'
{
  "query": {
    "filtered": {
      "query": {
        "bool": {
          "must": [
            {
              "multi_match": {
                "query": "australian system",
                "fields": [
                  "title^100",
                  "text^10",
                  "_all"
                ]
              }
            }
          ]
        }
      },
      "should": [
        {
          "match_phrase": {
            "title": {
              "query": "australian system",
              "slop": 1
            }
          }
        },
        {
          "match_phrase": {
            "text": {
              "query": "australian system",
              "slop": 1
            }
          }
        }
      ]
    }
  }
}
```

```

    "filter": {
      "bool": {
        "must_not": [
          {
            "term": {
              "redirect": "true"
            }
          },
          {
            "term": {
              "special": "true"
            }
          }
        ]
      }
    }
  }
}

```

命令的响应结果如下：

```

Australian honours system
Australian Series System
Australian soccer league system
Australian Antarctic Building System
Australian Integrated Forecast System
Australian Defence Air Traffic System
Western Australian Land Information System
The Australian Advanced Air Traffic System
Australian archaeology
Australian Democrats

```

结果变得更好了，不是吗？事实上还可以继续提高查询相关性。

5. 现在引入 boost

如果需要调整短语查询的权重，可以用 `function_score` 查询把短语查询包装一下。例如，假如我们想将一个针对 `title` 字段的短语查询的权重设置为 1000，可以调整之前查询的下面这个部分：

```

...
{
  "match_phrase": {
    "title": {
      "query": "australian system",
      "slop": 1
    }
  }
}

```

```

    }
  }
  ...

```

它将被调整为以下内容:

```

...
{
  "function_score": {
    "boost_factor": 1000,
    "query": {
      "match_phrase": {
        "title": {
          "query": "australian system",
          "slop": 1
        }
      }
    }
  }
}
...

```

在经过调整后, 带有短语的文档的得分将高于之前的得分, 但这留给读者去测试。

6. 创建一个可纠正拼写错误的查询系统

回头看一下索引的映射配置, 你将发现有一个 title 字段被定义为 multi_field 类型, 而其中的一个字段需要被 ngram 分析器做分词处理。默认情况下, ngram 分析器会产生 bigram, 例如, 对于单词 system, 将生成 sy ys st te em 等几个 bigram。想象一下, 我们可以在查询时忽略其中的一些分词结果, 从而使我们的查询系统具备自动纠正拼写错误的能力。我们用一个简单的查询来演示具体该怎么做:

```

curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d'
{
  "query": {
    "query_string": {
      "query": "austrelia",
      "default_field": "title",
      "minimum_should_match": "100%"
    }
  }
}'

```

返回的查询结果如下:

```

{
  "took" : 10,
  "timed_out" : false,
  "_shards" : {

```

```

    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 0,
    "max_score" : null,
    "hits" : [ ]
  }
}

```

在这里我们针对 title 字段发送了一个带有错误拼写的查询命令。因为索引中没有和拼写错误词项完全匹配的文档，我们什么也没得到。然后我们转而使用 title.ngram 字段，并忽略一些 bigram，这样 Elasticsearch 就可以找到一些匹配的文档了。修改后的查询命令如下：

```

curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d'
{
  "query": {
    "query_string": {
      "query": "austrelia",
      "default_field": "title.ngram",
      "minimum_should_match": "85%"
    }
  }
}'

```

在这里我们把 default_field 属性由 title 改为 title.ngram，从而让 Elasticsearch 使用会生成 bigram 的字段。另外，我们还引入了 minimum_should_match，把它设置为 85%。这个设置的意思是让 Elasticsearch 知道，我们并不想要所有的 bigram 分词结果都匹配上，而是匹配它们中的一部分，我们也不关心具体哪些词项会被匹配。



注意 调小 minimum_should_match 的取值，我们将得到更多的文档，不过会降低精度。相反，调大它的取值，我们将得到较少的文档，不过返回的文档中会包含更多匹配到的 bigram 分词结果，因此相关性更高。

查询结果的前几条如下：

```

Aurelia (Australia)
Australian Kestrel
Austrlia
Australian-Austrian relations
Australia-Austria relations
Australia-Austria relations
Australian religion
CARE Australia

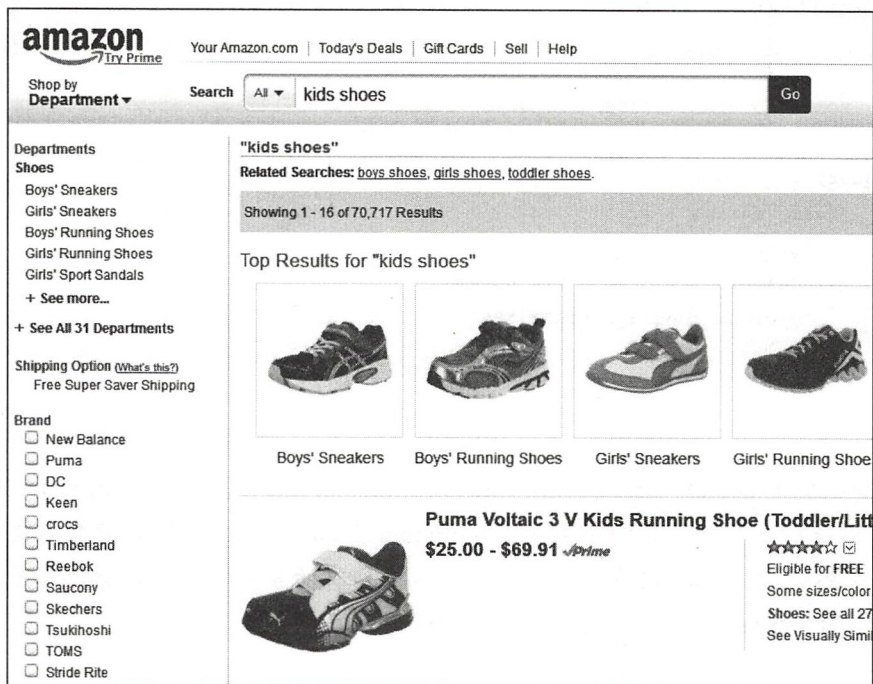
```


Care Australia
Felix Austria

如果你想要了解使用 Elasticsearch 的 suggester 来做拼写检查的相关知识，请查阅 4.1 节。

7. 继续探讨切面计算

最后我们想要讨论的话题是切面计算（faceting）。你可用切面计算同时做好几件事，比如计算直方图、字段统计、地理距离范围等。不过，只有词项切面计算（terms faceting）能够切实帮助用户获取想要的结果。例如，在 amazon.com 的搜索框中输入“kids shoes”，你将看到类似下面的截图：



你可以通过选择页面左侧的品牌来缩小结果范围。品牌列表不是静态的，而是基于搜索结果动态生成的。我们也可以通过 Elasticsearch 的词项切面计算达到同样的效果。



注意 请记住，我们同时使用切面计算及聚合计算来演示查询。切面计算以及被废弃，将在之后被从 Elasticsearch 中移除。尽管如此，由于有不少用户一直在使用切面计算，我们还是使用同一个查询的不同变体来演示范例。

让我们回到之前的 wikipedia 数据。假定用户可以在初次查询后通过选择文档分类来缩小结果范围。为了达到这一目的，我们需要在查询中加入 facets 设置（为了简化示例，我们使用 match_all 查询替换之前的复杂查询），新的查询命令如下：

```
curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d '{
  "query": {
    "match_all": {}
  },
  "facets": {
    "category_facet": {
      "terms": {
        "field": "category.untouched",
        "size": 10
      }
    }
  }
}'
```

在这里，词项切面计算基于已经索引的数据。我们选择在 `category.untouched` 字段上执行切面计算。如果选择在 `category` 字段上执行切面计算，则结果中只有一个单独的词项，而我们想要完整的分类。查询结果的切面计算结果部分如下（完整内容保存在随本书一起提供的 `response_query_facets.json` 文件中）：

```
"facets" : {
  "category_facet" : {
    "_type" : "terms",
    "missing" : 6175806,
    "total" : 16732022,
    "other" : 16091291,
    "terms" : [ {
      "term" : "Living people",
      "count" : 483501
    }, {
      "term" : "Year of birth missing (living people)",
      "count" : 39413
    }, {
      "term" : "English-language films",
      "count" : 22917
    }, {
      "term" : "American films",
      "count" : 16139
    }, {
      "term" : "Year of birth unknown",
      "count" : 15561
    }, {
      "term" : "The Football League players",
      "count" : 14020
    }, {
      "term" : "Main Belt asteroids",
      "count" : 13968
    }, {
      "term" : "Black-and-white films",
```

```

    "count" : 12945
  }, {
    "term" : "Year of birth missing",
    "count" : 12442
  }, {
    "term" : "English footballers",
    "count" : 9825
  } ]
}
}

```

默认情况下，切面计算结果按 count 属性降序排列。count 属性代表特定类别的结果数量。当然，我们也可以使用如下基于聚合的查询：

```

curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d '{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "category_agg": {
      "terms": {
        "field": "category.untouched",
        "size": 10
      }
    }
  }
}'

```

现在，如果我们的用户想将返回结果缩窄至 English-language 电影类别，可使用下面这个查询：

```

curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d '{
  "query": {
    "filtered": {
      "query" : {
        "match_all" : {}
      },
      "filter" : {
        "term": {
          "category.untouched": "English-language films"
        }
      }
    }
  },
  "facets": {

```

```

    "category_facet": {
      "terms": {
        "field": "category.untouched",
        "size": 10
      }
    }
  }
}

```

我们修改了查询，使之包含了一个过滤器，过滤器过滤的文档将会做切面计算处理。当然，也可通过下面这个聚合查询来实现同样的目的：

```

curl -XGET 'localhost:9200/wikipedia/_search?fields=title&pretty' -d '{
  "query": {
    "filtered": {
      "query" : {
        "match_all" : {}
      },
      "filter" : {
        "term": {
          "category.untouched": "English-language films"
        }
      }
    }
  },
  "aggs": {
    "category_agg": {
      "terms": {
        "field": "category.untouched",
        "size": 10
      }
    }
  }
}'

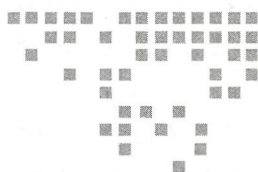
```

4.3 小结

本章我们学习了如何通过 term suggerter 和 phrase suggerter 来纠正用户的拼写错误。因此，我们已经掌握了避免因拼写错误产生空白页的技巧。另外，我们通过改进查询相关度来改善了用户搜索体验。我们从一个简单查询开始，然后逐步添加了多匹配查询、短语查询、查询权重 (boost)、查询间距 (query slop) 等功能。我们还展示了如何过滤掉垃圾结果，

以及如何优化短语匹配的权重设置。我们使用 `n-grams` 分词代替 Elasticsearch 的 `suggester` 来避免拼写错误。我们还探讨了使用切面计算来缩小搜索结果范围，从而简化用户找到想要文档或产品的途径。

下一章我们将探讨与性能相关的话题。最开始，我们会讨论 Elasticsearch 中的可扩展性。之后将了解如何为集群部署选择合适的分片数及副本数，以及剖析路由（`routing`）是如何帮助系统部署的。紧接着将研究如何更改默认的分片分配逻辑以满足应用需求。最后，将了解 Elasticsearch 提供了哪些查询执行逻辑，以及如何控制它来完美地适应我们的部署目标及索引架构。



分布式索引架构

在上一章，我们聚焦于改善用户搜索体验。我们首先使用了词项和短语建议器来改正用户查询中的拼写错误，然后使用自动完成建议器实现了高效的、基于索引的自动完成功能。最后，我们还了解了 Elasticsearch 调优的一些知识。我们以一个简单查询开始，逐渐添加多匹配查询、短语查询、调整权重以及查询间隔等功能。我们学习了如何过滤掉垃圾结果，以及如何改善短语查询的效果。我们使用 `n-grams` 来避免拼写错误，并把它作为 Elasticsearch 查询建议器的一种替代方法。我们还探讨了如何使用分组查询来筛选查询结果，从而简化用户找出期望文档或产品的途径。到本章结束时，将涵盖以下内容：

- 选择合适的分片数和副本数
- 路由
- 分片分配行为的微调
- 利用查询执行偏好

5.1 选择合适的分片和副本数

当你开始使用 Elasticsearch 时，大概会先创建索引，往索引中导入数据，然后开始执行查询。我们很确定开始的时候一切都很顺利，至少在数据量不大和查询压力不是很高的时候。在后台，Elasticsearch 会创建一些分片和适当数量的副本（如果使用默认配置的话），一般情况下你也不会部署时过多关注这部分的配置。

随着应用程序的成长，你不得不索引越来越多的数据，每秒钟处理越来越多的请求。

这个时候一切都变了，问题开始出现了（可阅读第 8 章的内容来了解如何应对应用程序的扩张）。现在应该考虑如何规划你的索引及其配置，使它们适应应用的变化。在本章里，我们将给出如何处理这个问题的一些指导方针。不幸的是没有确切的秘诀，每个应用程序都有各自的特性和需求，不仅索引结构依赖于此，配置也依赖于此。例如，文档或者索引的大小、查询类型以及期望的吞吐量都是其影响因素。

5.1.1 分片和过度分配

读者已经在 1.2 节中了解了索引分片的相关概念，不过在这里我们还是来回忆一下。分片是将一个 Elasticsearch 索引分割成一些更小的索引的过程，我们能够在同一集群的不同节点上散布它们。在查询的时候，结果汇总了索引中每个分片的返回结果（有时可能不是真的汇总，因为可能某个分片上就包含了所有感兴趣的数据）。Elasticsearch 默认为每个索引创建 5 个分片，即使在单节点环境下也是如此。这种冗余被称作过度分配（over allocation）。目前看起来这么做是完全没有必要的，反而在索引文档（散布文档到各个分片）和处理查询（查询多个分片合并结果）时增加了复杂性。幸运的是，这种复杂性被 Elasticsearch 自动处理了。既然如此，Elasticsearch 为什么还要这么做呢？

例如说我们有一个且仅有一个分片的索引。这意味着当应用程序的增长超过了单一服务器的容量时，我们会遇到问题。当前的 Elasticsearch 版本还无法将索引分割成多份，我们必须在创建索引时就指定好需要的分片数量。我们所能做的只有创建一个拥有更多分片的新索引，并重新索引数据。然而，这样的操作需要额外的时间和服务器资源，例如 CPU、内存和大量的存储。我们可能根本就没有前面提到的时间和资源。另一方面，当使用过度分配时，我们可以仅仅增加一台安装了 Elasticsearch 的服务器，Elasticsearch 会重新平衡（rebalance）集群，将部分索引迁移到新的机器上，不需要额外的重新索引数据的开销。Elasticsearch 设计者选择的默认配置（5 个分片和 1 个副本）在数据量增长和多分片搜索结果合并之间做了平衡。

默认的 5 个分片是标准用法。那么问题就来了：什么时候我们需要用更多的分片，或者与之相反，使用尽可能少的分片数量？

第一个答案很明显。如果你有一个有限的和明确的数据集，你可以只使用一个分片。如果没有，那么依照经验，最理想的分片数量应该依赖于节点的数量。因此，如果你计划将来使用 10 个节点，你需要给索引配置 10 个分片。需要记住的一点是：为了保证高可用和查询的吞吐量，我们同样需要配置副本数，而且它跟普通的分片一样需要占用节点上的空间。如果每个分片有一份额外的拷贝（`number_of_replicas` 等于 1），你最终会有 20 个分片。10 个包含主数据，10 个是其副本。

总的来说，节点数和分片数、副本数的简单计算公式如下：

所需最大节点数 = 分片数 * (副本数 + 1)

换句话说, 如果你计划使用 10 个分片和 2 个副本, 那么所需的最大的节点数是 30。

5.1.2 一个过度分配的正面例子

如果你仔细阅读了本章前面的部分, 你就会有强烈的信念: 我们应该使用最少数量的分片。但是有时拥有更多的分片有其便利之处, 因为一个分片事实上是一个 Lucene 索引。更多的分片意味着每个在较小的 Lucene 索引上执行的操作会更快 (尤其是索引过程)。有时这是一个使用更多分片的很好的理由。当然, 将查询分散成对每个分片的请求, 然后合并结果, 这也是有代价的。这个对于使用固定的参数来过滤查询的应用程序是可以避免的。有这种现实的案例, 例如那种每个查询都在指定用户的上下文中执行的多租户系统。原理很简单, 我们可以将每个用户的数据都索引到一个独立的分片中, 在查询时就只查询那个用户的分片。这时需要使用路由 (我们将在 5.2 节中详细讨论它)。

5.1.3 多分片与多索引

你可能会奇怪, 如果一个分片事实上是一个小的 Lucene 索引, 那么什么才是真正的 Elasticsearch 索引? 拥有多个分片和拥有多个索引有什么不同? 从技术上讲, 它们的区别不大, 而且对于某些应用场景来说, 使用多个索引是更好的选择 (例如, 将基于时间的数据如日志等索引到以时间段切分的不同索引中)。如果使用拥有多个分片的单个索引, 某些时候可以通过路由把查询定位到一个分片上。而如果使用多个索引, 你可以有机会选择只在那些感兴趣的索引上执行查询, 比如, 通过名称为 logs_2014-10-10、logs_2014-10-11……这样的方式来选择基于时间区间构建的索引。更多的不同可以在分片和索引的平衡逻辑上看起来, 尽管可以人为配置两者的平衡逻辑。

5.1.4 副本

分片处理使我们能存储超过单节点容量的数据, 而使用副本则解决了日渐增长的吞吐量和数据安全方面的问题。当一个存放主分片的节点失效后, Elasticsearch 能够升级一个可用的副本为新的主分片。默认情况下, Elasticsearch 只为每个索引分片创建一个副本。然而, 不同于分片处理, 副本的数量可以通过使用相关 API 随时更改。该功能让构建应用程序变得非常方便, 因为我们的查询吞吐量随着用户的增长而增长, 而使用副本则可以应对增长的并发查询。增加副本数目可以让查询负载分散到更多机器上, 理论上可以让支持处理更多的并发请求。

使用过多副本的缺点也很明显: 额外副本占用了额外的存储空间, 构建索引副本的开销。当然, 主分片及其副本之间的数据拷贝也存在开销。在选择分片数量的时候你应当同

时考虑所需要的副本数量。如果选择了太多的副本，可能会耗光磁盘空间和 Elasticsearch 的资源，而事实上这些副本很多时候根本不会用到。另一方面，如果不创建副本，当主分片发生问题时，可能会造成数据的丢失。

5.2 路由

在 5.1 节中，我们提到过路由是限定查询在单个分片上执行的一个解决方案。现在是时候进一步介绍该功能了。

5.2.1 分片和数据

通常情况下，Elasticsearch 将数据分发到哪个分片，以及哪个分片上存放特定的文档是不重要的。查询时，请求会被发送至所有的分片，所以最关键的事情就是使用一个能均匀分发数据的算法，使得每个分片都包含差不多数量的文档。我们并不希望某个分片持有 99% 的数据，而另一个分片持有剩下的 1%，这样做极其低效。

而当我们想删除文档或者增加一个文档的新版本时，情况就有些复杂了。Elasticsearch 必须确定哪个分片需要更新。尽管看起来挺麻烦，实际上，这并不是一个大问题。只要分片算法能对同一个文档标识符永远生成相同的值就足够了。如果我们有了这样一个算法，Elasticsearch 在处理一个文档时就知道该去找哪个分片了。

另外，某些时候我们希望把数据的一部分索引到相同的分片上。举例来说，我们希望将特定类别的书籍都存在一个特定的分片上，在查询这类书时我们可以避免查询多个分片及合并搜索结果。这时候，因为我们确切地知道路由时使用的取值，就可以把 Elasticsearch 引导到与索引时相同的分片上。这就是路由要做的事情。它允许我们提供信息给 Elasticsearch，然后 Elasticsearch 用这个信息来决定哪个分片用来存储文档和执行查询。相同的路由值总是指向同一个分片。换个说法就是：“之前你使用某个路由值将文档存放在特定的分片上，那么搜索时，也去相应的分片查找该文档。”

5.2.2 测试路由功能

现在向读者展示一个例子，用来演示 Elasticsearch 是如何分配分片，以及如何将文档存放到特定的分片上。在这里，我们将使用一个额外的插件，名为 paramedic，它能帮助我们查看 Elasticsearch 到底对数据做了什么。可使用下面的命令来安装 paramedic 插件：

```
bin/plugin -install karumi/elasticsearch-paramedic
```

重启 Elasticsearch 后，可以通过浏览器访问 http://localhost:9200/_plugin/paramedic/index.html，然后会看到一个页面，上面有索引相关的各种统计量和其他信息。对于我们的

例子来说，最令人感兴趣的信息是象征集群状态的集群颜色和每个索引的分片和副本列表。

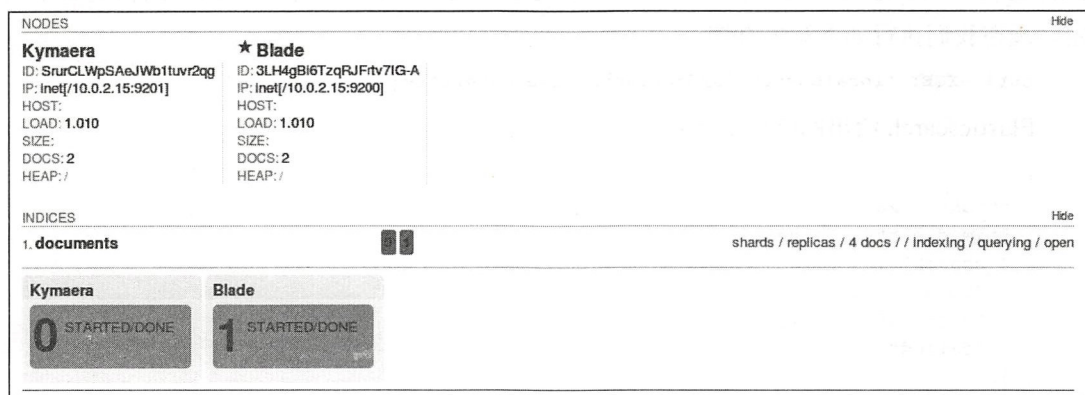
现在启动两个 Elasticsearch 节点，然后用下面的命令创建一个索引：

```
curl -XPUT 'localhost:9200/documents' -d '{
  "settings": {
    "number_of_replicas": 0,
    "number_of_shards": 2
  }
}'
```

我们创建了一个只有两个分片但是没有副本的索引。这意味着集群最多可以有两个节点，接下来的节点将不能填充数据，除非我们提高副本的数量（可参考 5.1 节）。下一步操作是索引文档，我们使用下面一系列命令：

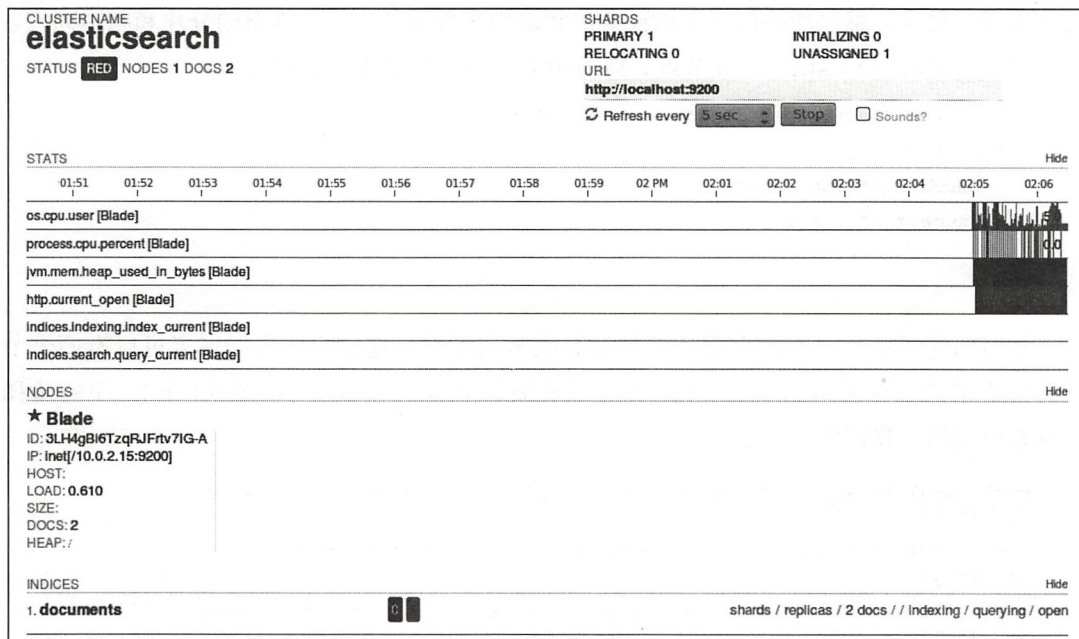
```
curl -XPUT localhost:9200/documents/doc/1 -d '{"title": "Document No. 1"}'
curl -XPUT localhost:9200/documents/doc/2 -d '{"title": "Document No. 2"}'
curl -XPUT localhost:9200/documents/doc/3 -d '{"title": "Document No. 3"}'
curl -XPUT localhost:9200/documents/doc/4 -d '{"title": "Document No. 4"}'
```

之后 Paramedic 向我们展示了两个主分片，见下面的截图：



在上面给出的节点相关的信息中，我们也找到了目前感兴趣的内容。集群中的每个节点都精确地容纳了两个文档，由此我们可以得出以下结论：分片算法完美地完成了它的工作，我们得到了一个含有多个分片的索引，文档在分片之间均匀分布。

现在让我们人为制造一些灾难：关闭第 2 个节点。现在使用 Paramedic 我们将看到类似下面的截图：



我们看到的第一个信息是集群的状态是红色的了。这意味着至少一个主分片丢失了，因此一些数据不再可用，索引的某些部分不再可用。尽管如此，Elasticsearch 还是允许我们执行查询。至于是通知用户查询结果可能不完整还是拒绝查询请求，则由应用构建者来决定。现在我们执行如下简单查询：

```
curl -XGET 'localhost:9200/documents/_search?pretty'
```

Elasticsearch 给出的响应如下：

```
{
  "took" : 26,
  "timed_out" : false,
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "documents",
      "_type" : "doc",
      "_id" : "2",
      "_score" : 1.0,
      "_source":{ "title" : "Document No. 2" }
    }, {
      "_index" : "documents",
```

```

    "_type" : "doc",
    "_id" : "4",
    "_score" : 1.0,
    "_source":{ "title" : "Document No. 4" }
  } ]
}

```

正如你所看到的，Elasticsearch 返回了关于故障的信息。我们看到有一个分片是不可用的。在返回的结果集中，我们只能看到标识符为 2 和 4 的文档。而其他文档丢失了，至少在主分片恢复正常之前是这样的。如果你启动第 2 个节点，经过一段时间（取决于网络和网关模块的设置）后，集群会恢复到绿色状态，此时所有的文档都可用。现在我们使用路由重复前面的范例，同时观察 Elasticsearch 的行为与上次有何不同。

在索引过程中使用路由

我们可以通过路由控制 Elasticsearch 选择将文档发送到哪个分片。此时需要指定路由参数 “routing”。路由参数值无关紧要，你可以选择任何值。重要的是在将不同文档放到同一个分片上时，需要使用相同的值。简单地说，给不同的文档使用相同路由参数值将确保这些文档被索引到相同分片中。

向 Elasticsearch 提供路由信息有多种途径。最简单的办法是在索引文档时加一个 URI 参数 routing。例如下面的例子：

```

curl -XPUT localhost:9200/books/doc/1?routing=A -d '{ "title" :
"Document" }'

```

当然，我们也可以在批量索引时使用这个参数。在批量索引时，路由参数由每个文档的元数据中的 “_routing” 属性指定。例如：

```

curl -XPUT localhost:9200/_bulk --data-binary '
{ "index" : { "_index" : "books", "_type" : "doc", "_routing" : "A"
}}
{ "title" : "Document" }
,

```

另一个选择是在文档里放一个 “_routing” 字段。不过，这种方式要想生效，需要确保先在索引映射中定义了 “_routing” 字段。例如，使用如下命令创建一个命名为 “books_routing” 的索引：

```

curl -XPUT 'localhost:9200/books_routing' -d '{
  "mappings": {
    "doc": {
      "_routing": {
        "required": true,
        "path": "_routing"
      }
    }
  }
}

```



```

    },
    "properties": {
      "title" : { "type": "string" }
    }
  }
}
},

```

现在，可以在文档中使用“_routing”字段了。例如：

```

curl -XPUT localhost:9200/books_routing/doc/1 -d '{ "title" :
"Document", "_routing" : "A" }'

```

在这个例子里我们使用了_routing 字段。值得一提的是 path 参数可以指向文档中任意未分词字段。这是一个十分强大的功能，也是路由特性最主要的优势之一。举例来说，如果我们用代表图书所在图书馆的 library_id 字段扩展我们的文档，那么当基于 library_id 来设置路由时，有理由认为所有基于图书馆的查询更有效率。不过，需要注意，从文档字段中解析出“_routing”值需要额外的一些解析工作。

5.2.3 索引时使用路由

现在重复前面的例子，只是这次会使用路由。首先要删除旧文档。如果不这么做，那么使用相同的标识符添加文档时，路由会造成相同的文档被存放到另一个分片上去。因此，我们执行下面的命令从索引中删除所有的文档：

```

curl -XDELETE 'localhost:9200/documents/_query?q=*:*'

```

然后重新索引数据，但是这次添加路由信息进去。索引文档的命令如下：

```

curl -XPUT localhost:9200/documents/doc/1?routing=A -d '{ "title" :
"Document No. 1" }'
curl -XPUT localhost:9200/documents/doc/2?routing=B -d '{ "title" :
"Document No. 2" }'
curl -XPUT localhost:9200/documents/doc/3?routing=A -d '{ "title" :
"Document No. 3" }'
curl -XPUT localhost:9200/documents/doc/4?routing=A -d '{ "title" :
"Document No. 4" }'

```

路由参数指示 Elasticsearch 应该将文档放到哪个分片上。当然，同一个分片常常会存放多个文档。这是因为分片数往往少于路由参数值的个数。现在我们停掉一个节点，Paramedic 会再次显示红色的集群状态。执行匹配所有文档的查询，Elasticsearch 将返回相应的结果（当然，返回结果取决于我们停掉了哪个节点）：

```

curl -XGET 'localhost:9200/documents/_search?q=*&pretty'

```

Elasticsearch 的响应结果如下：

```

{
  "took" : 24,
  "timed_out" : false,
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "hits" : {
    "total" : 3,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "documents",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 1.0,
      "_source":{ "title" : "Document No. 1" }
    }, {
      "_index" : "documents",
      "_type" : "doc",
      "_id" : "3",
      "_score" : 1.0,
      "_source":{ "title" : "Document No. 3" }
    }, {
      "_index" : "documents",
      "_type" : "doc",
      "_id" : "4",
      "_score" : 1.0,
      "_source":{ "title" : "Document No. 4" }
    } ]
  }
}

```

在这个例子里，标识符为 2 的文档不见了。我们失去了拥有路由值 B 的文档所在的节点。如果更倒霉一点，将会丢失 3 个文档！

查询

路由允许用户指定 Elasticsearch 应该在哪个分片上执行查询。既然我们只需要从索引的一个特定子集中获取数据，还有什么必要把查询发送到所有的节点呢？举例来说，假如我们想要从由路由值 A 确定的分片上查询数据，可以执行如下查询命令：

```
curl -XGET 'localhost:9200/documents/_search?pretty&q=*&routing=A'
```

我们仅仅在路由参数里加入了一个感兴趣的值。Elasticsearch 给出了下面的响应：

```

{
  "took" : 0,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,

```

```

    "failed" : 0
  },
  "hits" : {
    "total" : 3,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "documents",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 1.0, "_source" : { "title" : "Document No. 1" }
    }, {
      "_index" : "documents",
      "_type" : "doc",
      "_id" : "3",
      "_score" : 1.0, "_source" : { "title" : "Document No. 3" }
    }, {
      "_index" : "documents",
      "_type" : "doc",
      "_id" : "4",
      "_score" : 1.0, "_source" : { "title" : "Document No. 4" }
    } ]
  }
}

```

一切都工作的似乎很好，但是仔细看看！我们忘记启动一个节点，这个节点上的分片包含着索引时使用了路由值 B 的文档。尽管我们没有一个完整的索引视图，Elasticsearch 的响应中并未包含分片失败的信息。这证明了使用路由的查询只命中选定的分片，忽略其他分片。如果我们使用路由值 B 再执行一次查询，我们会得到类似下面的异常：

```

{
  "error" : "SearchPhaseExecutionException[Failed to execute phase
[query_fetch], all shards failed]",
  "status" : 503
}

```

我们可以通过 Search Shard API 来验证刚才的行为。例如，执行如下命令：

```

curl -XGET 'localhost:9200/documents/_search_shards?pretty&routing=A'
-d '{"query":"match_all":{"}}'

```

Elasticsearch 的响应如下：

```

{
  "nodes" : {
    "QK5r_d5CSfaVlWx78k633w" : {
      "name" : "Western Kid",
      "transport_address" : "inet[/10.0.2.15:9301]"
    }
  },
  "shards" : [ [ {
    "state" : "STARTED",
    "primary" : true,

```

```

    "node" : "QK5r_d5CSfaVlWx78k633w",
    "relocating_node" : null,
    "shard" : 0,
    "index" : "documents"
  } ] ]
}

```

从响应中可以看出，只有一个节点被查询了。

有件重要的事情需要再强调一下。路由确保在索引时拥有相同路由值的文档被索引到相同的分片上。但是，你需要记住一个给定的分片上可以有很多拥有不同路由值的文档。路由可以限制查询时使用的节点数，但是不能替代过滤功能。这意味着无论一个查询有没有使用路由都应该使用相同的过滤器。举个例子，如果我们拿用户标识来作为路由值，在搜索该用户的数据时，还应当在查询中包含一个按用户标识进行过滤的过滤器。

5.2.4 别名

如果你是一个搜索引擎专家，你大概会希望对程序员隐藏一些配置信息，好让程序员们可以快速地工作且不必关心搜索细节。在理想的世界里，他们不需要担心路由、分片、副本。别名让我们像使用普通索引那样来使用路由。例如，让我们用下面的命令来创建一个别名：

```

curl -XPOST 'http://localhost:9200/_aliases' -d '{
  "actions" : [
    {
      "add" : {
        "index" : "documents",
        "alias" : "documentsA",
        "routing" : "A"
      }
    }
  ]
}'

```

在前面的例子里我们创建了一个叫 documentsA 的虚拟索引（一个别名），用来代表来自 documents 索引的信息。同时，查询被限定在路由值 A 相关的分片上。多亏这个功能，你可以将别名 documentsA 的信息提供给开发者，他们可以直接用它进行查询和索引，就像其他索引一样。

5.2.5 多个路由值

Elasticsearch 允许我们在一次查询中使用多个路由值。文档被放置在哪个分片上依赖于

文档的路由值，多路由值查询意味着在一个或多个分片上查询。我们看看下面的查询：

```
curl -XGET 'localhost:9200/documents/_search?routing=A,B'
```

查询执行后，Elasticsearch 会将查询请求发送到两个分片上（在我们使用的示例中，刚好是索引的所有分片），这是因为路由值 A 涵盖了索引两个分片中的一个，而路由值 B 涵盖了另一个。

当然，多路由值也支持别名。下面的例子展示了如何使用这些特性：

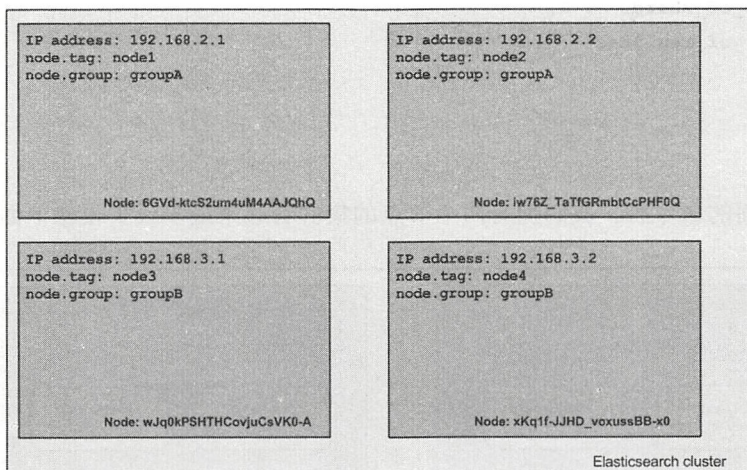
```
curl -XPOST 'http://localhost:9200/_aliases' -d '{
  "actions" : [
    {
      "add" : {
        "index" : "documents",
        "alias" : "documentsA",
        "search_routing" : "A,B",
        "index_routing" : "A"
      }
    }
  ]
}
```

上面的例子里有两个额外的配置参数是我们之前没有提到过的，我们可以为查询和索引配置不同的路由。在前面的例子里，我们定义在查询时（search_routing 参数）使用 2 个路由值（A 和 B）。而索引时（index_routing 参数）仅有一个路由值（A）被使用。记住，索引时不支持多个路由值，同时你也应记住要做适当的过滤（你也可以把它加到别名中）。

5.3 调整默认分片的分配行为

在 Packt 出版社出版的《Elasticsearch Server, Second Edition》一书中我们介绍了许多有关 Elasticsearch 分片分配功能的知识。我们探讨了 Cluster Reroute API，分片再平衡，以及分片部署意识。尽管这些知识不常用，但是如果你想完全掌控你的 Elasticsearch 集群，这些内容将非常重要。有鉴于此，我们决定扩展《Elasticsearch Server, Second Edition》一书中的相关示例，展示给读者关于以下主题的指导方针：如何使用 Elasticsearch 的分片部署意识功能，以及如何调整默认分片分配机制。

让我们从一个简单示例开始展开。假如我们有一个 4 个节点的集群，看上去像下面这样：



如你所见，我们的集群由 4 个节点构成。每个节点都绑定了一个指定的 IP 地址，每个节点都被赋予了一个 tag 属性和一个 group 属性（在 Elasticsearch.yml 文件里对应的是 node.tag 和 node.group 属性）。这个集群用来展示分片分配的过滤处理是如何工作的。你可以给 tag 和 group 属性任意名字，你只需要给你期望的属性名前面加上 node 前缀。例如，如果你想将 party 作为属性名，你只需要把 node.party: party1 加入你的 Elasticsearch.yml 文件里即可。

5.3.1 部署意识

部署意识允许我们使用通用参数来配置分片和它们的副本的部署。为何证明部署意识是如何工作的，我们将使用我们的示例集群。为了让例子能够说明问题，我们需要在 Elasticsearch.yml 文件里加入下面的属性：

```
cluster.routing.allocation.awareness.attributes: group
```

这会告诉 Elasticsearch 使用 node.group 属性作为意识参数。



注意 你可以指定多个值给 cluster.routing.allocation.awareness.attributes 属性，例如：

```
cluster.routing.allocation.awareness.attributes:
  group,
  node
```

然后，我们先启动前两个节点，node.group 属性值是 groupA 的那两个。接下来用下面的命令创建一个索引：

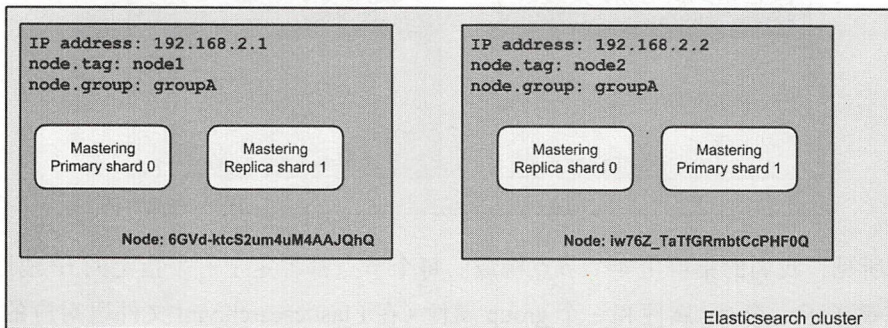
```
curl -XPOST 'localhost:9200/mastering' -d '{
  "settings" : {
    "index" : {
```

```

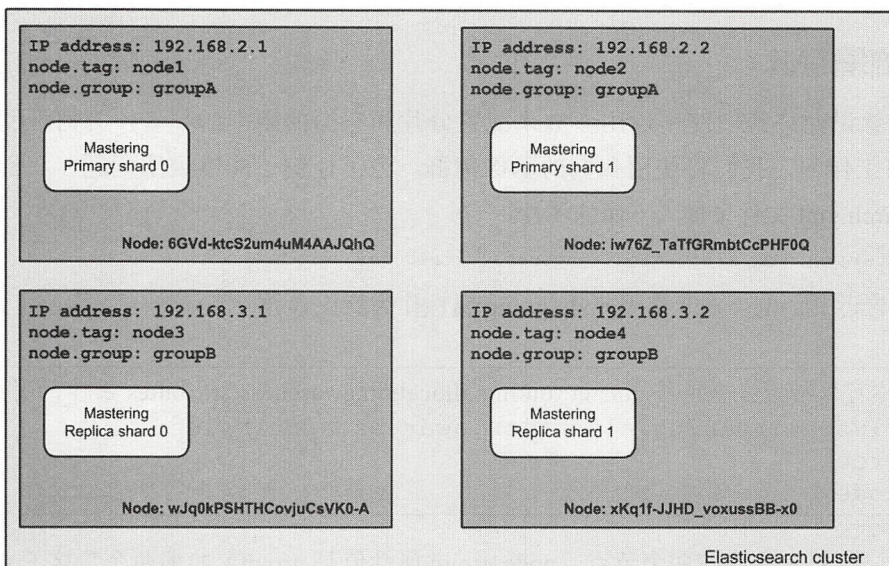
    "number_of_shards" : 2,
    "number_of_replicas" : 1
  }
}
}'

```

执行了前面的命令后，我们拥有两个节点的集群看起来或多或少很像下面的截图：



如你所见，索引被平均部署到了两个节点上。现在让我们看看当我们加入其余两个节点时会发生什么（node.group 设置成 groupB 的那两个）：



注意以下区别：主分片没有从原来部署的节点上移动，但是副本分片移动到了有不同 node.group 值的节点上。这恰恰是对的。当使用分片部署意识的时候，Elasticsearch 不会将主分片和副本放到拥有相同属性值（用来决定部署意识，在我们的例子是 node.gorup）的

节点上。使用这个功能的一个例子是从虚拟机或则物理位置的角度分割集群的拓扑结构，以确保你不会有单点故障。



注意 请记住，在使用部署意识的时候，分片不会被部署到没有设定指定属性的节点上。所以对我们的例子来说，一个没有设置 `node.group` 属性的节点是会被部署机制考虑的。

强制部署意识

在我们预先知道我们的意识参数需要接受几个值，且我们不希望超过我们需要的副本被部署到我们的集群里时（例如不想因过多的副本而使集群过载），有了强制部署意识就很方便了。为了实现这个，我们可以强制部署意识由特定属性激活。我们可以通过使用 `cluster.routing.allocation.awareness.force.zone.values` 属性，并给它提供一个用逗号分隔的列表值来指定这些属性值。例如，如果我们希望对于部署意识来说只使用 `node.group` 属性的 `groupA` 和 `groupB` 两个值，我们应该把下面的代码加到 `Elasticsearch.yml` 文件中：

```
cluster.routing.allocation.awareness.attributes: group
cluster.routing.allocation.awareness.force.zone.values: groupA,
groupB
```

5.3.2 过滤

Elasticsearch 允许我们在整个集群或是索引的级别来配置分片的分配。在集群的级别上我们可以使用带下面前缀的属性：

- `cluster.routing.allocation.include`
- `cluster.routing.allocation.require`
- `cluster.routing.allocation.exclude`

而处理索引级的分配时，使用带下面前缀的属性：

- `index.routing.allocation.include`
- `index.routing.allocation.require`
- `index.routing.allocation.exclude`

前面提到的前缀可以与我们在 `elasticsearch.yml` 文件里定义的属性一起使用（`tag` 属性和 `group` 属性）。通过使用一个名为 `_ip` 的特殊属性，我们就可以使用 `ip` 地址来进行包含或者排除特定的节点。例如：

```
cluster.routing.allocation.include._ip: 192.168.2.1
```

如果我们希望包含一组 `group` 属性是 `groupA` 的节点，我们应该设置下面的属性：

```
cluster.routing.allocation.include.group: groupA
```


请注意，我们已经使用了 `cluster.routing.allocation.include` 前缀，把它和属性名 `group` 连接在一起。

这些参数意味着什么

如果你仔细观察一下前面提到的参数，就会注意到它们有 3 种类型。

- `include`：这种类型包含所有定义了这个参数的节点。如果定义了多个 `include` 条件，那么至少匹配一个条件的节点都会在分配分片时被考虑进去。举例来说，如果我们增加 2 个 `cluster.routing.allocation.include.tag` 参数到我们的配置中，一个赋值 `node1`，另一个赋值 `node2`，结果就是索引（确切地说是它们的分片）被分配到了第 1 个和第 2 个节点上（从左向右）。总结一下，拥有 `include` 参数类型的节点，Elasticsearch 在选择放置分片的节点时会加以考虑，但是这并不意味着 Elasticsearch 一定会把分片放到这些节点上。
- `require`：这种类型是在 Elasticsearch 0.90 版本的分配过滤器中被引入的。它要求所有的节点都必须拥有和这个属性值相匹配的值。例如，如果我们向配置中添加 `cluster.routing.allocation.require.tag` 参数并赋值 `node1`，添加 `cluster.routing.allocation.require.group` 参数并赋值 `groupA`。结果就是所有的分片都分配在第一个节点上（IP 地址为 192.168.2.1 的节点）。
- `exclude`：这个属性允许我们在分片分配过程中排除具有特定属性的节点。例如，我们给 `cluster.routing.allocation.include.tag` 赋值 `groupA`，最终，索引只被分配在了 IP 地址是 192.168.2.1 和 192.168.2.2 的节点上（例子中第 3 和第 4 个节点）。



注意 属性值可以使用简单的通配符。例如，如果我们希望包含所有 `group` 属性值以 `group` 开头的节点，我们应当设置 `cluster.routing.allocation.include.group` 属性的值为 `group*`。就样例集群来说，这会导致匹配 `group` 参数值是 `groupA` 和 `groupB` 的节点。

5.3.3 运行时更新分配策略

除了在 `Elasticsearch.yml` 文件里设置我们讨论过的那些属性外，在集群已经启动运行后，我们也可以通过更新 API 来实时更新这些设置。

1. 索引级更新

为了更新一个给定索引（例如 `mastering` 索引）的设置，我们执行下面的命令：

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
  "index.routing.allocation.require.group": "groupA"
}'
```

如你所见，命令发送给指定索引的 `_settings` 端点。你可以在一次调用中包含多个属性。

2. 集群级更新

为了更新整个集群的设置，我们执行下面的命令：

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "transient" : {
    "cluster.routing.allocation.require.group": "groupA"
  }
}'
```

如你所见，命令被发送至 `_cluster/settings` 端点。你可以在一次调用中包含多个属性。注意前面命令中的 `transient`，它意味着在集群重启后属性将失效。如果想避免这样，想让属性持久化，可以用 `persistent` 属性替换 `transient` 属性。一个在重启后保留设置的例子如下：

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "persistent" : {
    "cluster.routing.allocation.require.group": "groupA"
  }
}'
```



注意 依赖于命令的内容和索引的分配情况，执行前面的命令可能造成分片在节点间的移动。

5.3.4 确定每个节点允许的总分片数

除了我们前面提到的属性，还能够定义每个节点上可以分配给一个索引的分片总数（主分片和副本）。为了实现该目的，我们需要给 `index.routing.allocation.total_shards_per_node` 属性设置一个期望值。例如在 `Elasticsearch.yml` 文件里我们设置：

```
index.routing.allocation.total_shards_per_node: 4
```

这会造成单个节点上最多为同一个索引分配 4 个分片。

这个属性同样可以在一个运行中的集群上使用 Update API 来改变，例如：

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
  "index.routing.allocation.total_shards_per_node": "4"
}'
```

现在，我们来看一些例子，在 `elasticsearch.yml` 文件中使用分片分配属性后，创建单一索引时集群是什么样子的。

5.3.5 确定每个物理机器允许的总分片数

当我们在单个物理机器上运行多个 Elasticsearch 节点时，有一个属性值得我们注意：

`cluster.routing.allocation.same_shard.host`。把这个属性设置为 `true` 将阻止 Elasticsearch 将主分片和它的副本部署在同一台物理主机上。如果你拥有性能强大的服务器，并且打算在一个物理主机上运行多个节点，我们强烈建议你设置这个属性的值为 `true`。

1. 包含

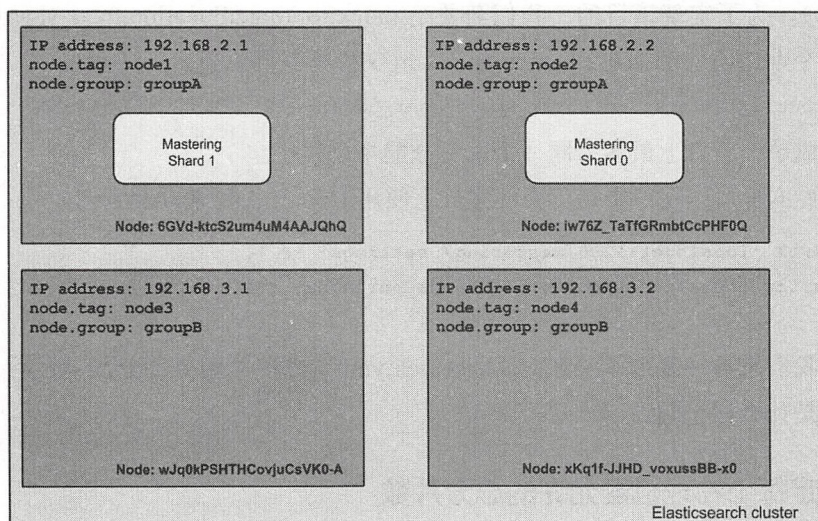
我们现在使用示例集群看看包含（include）方式是如何工作的。首先，用下面的命令删除并重新创建 `mastering` 索引：

```
curl -XDELETE 'localhost:9200/mastering'
curl -XPOST 'localhost:9200/mastering' -d '{
  "settings" : {
    "index" : {
      "number_of_shards" : 2,
      "number_of_replicas" : 0
    }
  }
}'
```

然后我们执行下面的命令：

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
  "index.routing.allocation.include.tag": "node1",
  "index.routing.allocation.include.group": "groupA",
  "index.routing.allocation.total_shards_per_node": 1
}'
```

如果我们把索引状态命令的响应可视化，就会看到集群看起来与下面的图片很相似：



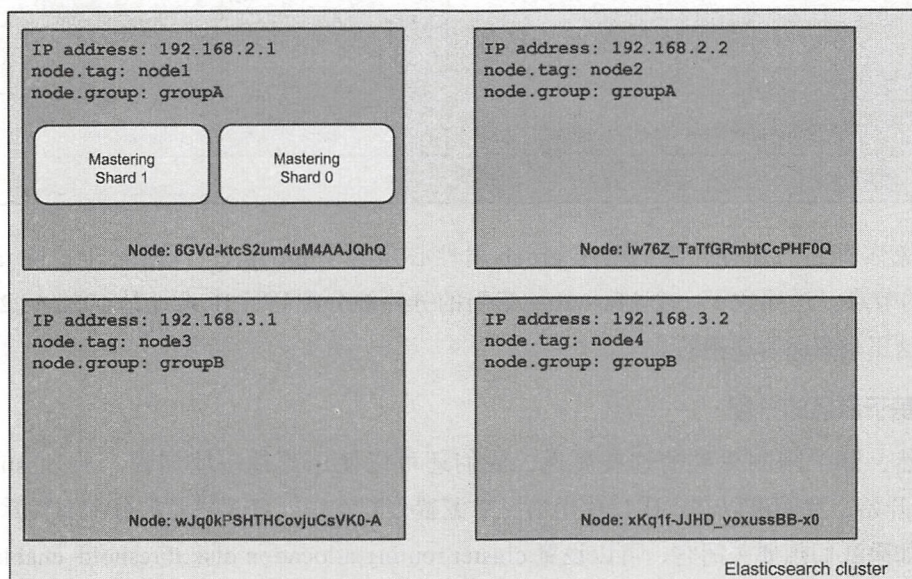
就像你看到的那样，Mastering 索引的分片被部署到了 tag 属性为 node1 或者 group 属性为 groupA 的节点上了。

2. 必须

现在我们还是使用前面的示例集群（假定集群里没有任何索引）来观察必须（require）方式是如何工作的。执行下面的命令：

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
  "index.routing.allocation.require.tag": "node1",
  "index.routing.allocation.require.group": "groupA"
}'
```

如果我们把索引状态命令的响应可视化，就会看到集群看起来像下面的样子：



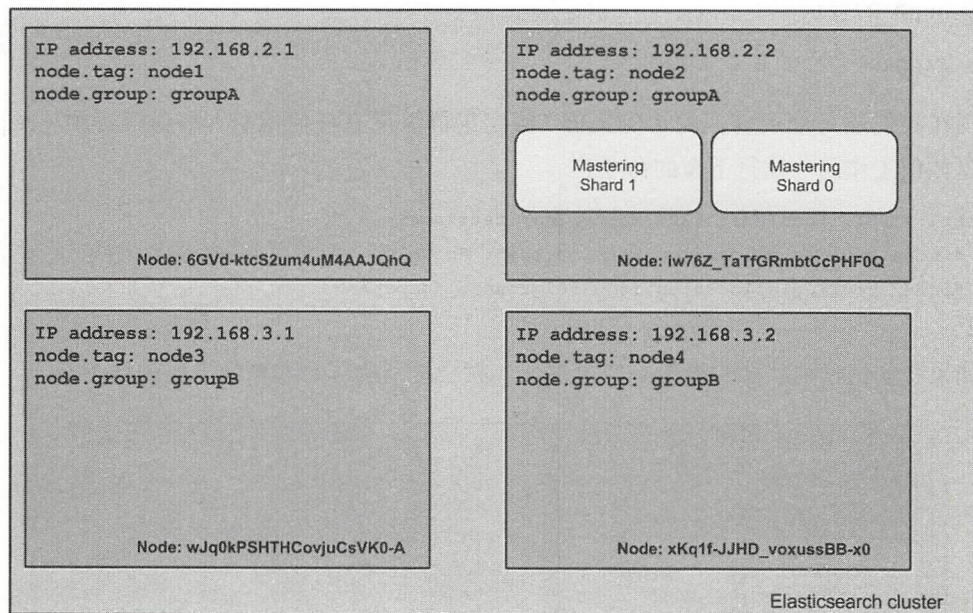
就像你看到的那样，这个视图跟前面我们使用 include 选项的那个不一样了。这是由于我们告诉 Elasticsearch，将 Mastering 索引的分片仅分配到与两个 require 参数都匹配的节点上，我们的例子里两个都匹配的只有第一个节点。

3. 排除

接着我们再看看排除（exclude）方式。执行下面的命令：

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
  "index.routing.allocation.exclude.tag": "node1",
  "index.routing.allocation.require.group": "groupA"
}'
```


再看看我们的集群：



就像你看到的那样，我们要求 group 属性必须等于 groupA，同时希望排除 tag 等于 node1 的节点。这导致了一个 Mastering 索引的分片被分配到了 IP 地址是 192.168.2.2 的节点上，这正是我们所期望的。

4. 基于磁盘的分配

当然，除了刚刚提到的这些属性，我们还可以使用其他一些属性。从 Elasticsearch 1.3.0 版开始，我们可以基于磁盘使用情况来配置分配意识。基于磁盘的分配属性默认是开启的。如果我们想要关闭它，可以设置 cluster.routing.allocation.disk.threshold_enabled 属性值为 false。

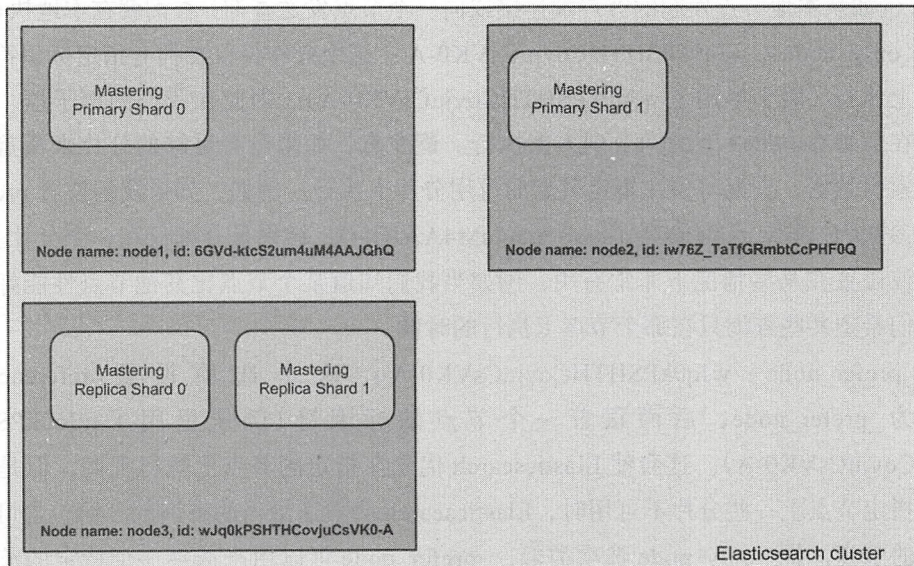
还有另外几个属性可以辅助我们配置基于磁盘的分片分配行为。第一个属性是 cluster.routing.allocation.disk.watermark.low，可以让 Elasticsearch 在触发条件时不再在节点上分配新的分片。该属性的默认值是 85，意味着当磁盘使用率达到 85% 后，节点上将不再分配新的分片。另一个属性是 cluster.routing.allocation.disk.watermark.high，这个属性可以让 Elasticsearch 在触发条件时尝试将分片从本节点上迁移出去。这个属性的默认值是 90，意味着当磁盘使用率达到 90% 后，Elasticsearch 将尝试将部分分片从本节点上迁出。

low 和 high 属性都可以设置为绝对值，比如，1024MB。

5.4 查询执行偏好

让我们忘记分片部署及其配置，至少就现在而言。除了 Elasticsearch 允许我们设置分片和副本的那些技巧以外，我们还能够指定查询（以及其他操作，例如实时获取）在哪里执行。

在了解细节之前，先查看一下我们的示例集群：



如你所见，集群中有 3 个节点和一个叫作 Mastering 的索引。索引被分割为两个主分片，每个主分片有一个副本。

介绍 preference 参数

为了控制我们发送的查询（和其他操作）执行的地点，可以使用 preference 参数，它可以被赋予下面这些值中的一个。

- ❑ `_primary`：使用这个属性，我们发送的操作仅在主分片上执行。所以如果我们向 mastering 索引发送一个查询请求，并将 preference 参数设置为 `_primary`。那么这个请求会在 node1 和 node2 上执行。例如，如果你知道你的主分片在某个机柜，副本在其他机柜，你可能希望通过在主分片上执行操作来避免网络开销。
- ❑ `_primary_first`：这个属性的行为很像 `_primary`，但它有一个自动故障恢复机制。如果我们向 mastering 索引发送了一个查询请求，并设置 preference 参数为 `_primary_first`，那么查询会在 node1 和 node2 上执行。而一旦一个（或多个）主分片失败了，查询会在另一个分片上执行，在我们的例子里这个分片位于 node3 上。就像我们说的，该选项非常像 `_primary`，只是当主分片由于某些原因不可用时转而使用其副本。

- ❑ `_local` : Elasticsearch 在可能的情况下会优先在本地节点上执行操作。例如, 如果我们发送一个查询请求给 node3 同时将 `preference` 参数设为 `_local`, 那么查询就会在该节点上执行。然而, 如果我们将查询发送给 node2, 那么就会有一个查询在主分片 1 上执行 (部署在节点 node2 上), 另一部分查询会在包含分片 0 的 node1 或 node3 上执行。这在我们想最小化网络延时尤其有用, 当使用 `_local` 时, 确保只要有可能 (例如从本地节点发起的客户端连接或向一个节点发送查询) 查询就在本地执行。
- ❑ `_only_node` : `wJq0kPSHTHCovjuCsVK0-A` : 这个操作将仅在拥有指定标识符的节点上执行 (例子里用了 `wJq0kPSHTHCovjuCsVK0-A`)。因此在我们的例子里, 查询会在部署在 node3 上的两个副本上执行。请注意, 如果没有足够的分片来覆盖所有的索引数据, 查询将仅在指定节点的可用分片上执行。例如, 如果我们设置 `preference` 参数为 `_only_node:6GVd-ktcS2um4uM4AAJQhQ`, 结果是查询仅在一个分片上执行。该设置在某些情况下非常有用, 例如当我们知道某个节点比其他节点性能好, 且我们希望某些查询只在那个节点上执行的时候。
- ❑ `_prefer_node` : `wJq0kPSHTHCovjuCsVK0-A` : 这个选项设置 `preference` 参数为 `_prefer_node`, 后面跟着一个节点的标识符 (例子里用了 `wJq0kPSHTHCovjuCsVK0-A`)。这会使 Elasticsearch 优先在指定的节点上执行查询, 但是如果该指定节点上一些分片不可用时, Elasticsearch 会发送恰当的查询部分给包含可用分片的节点。同 `_only_node` 选项类似, `_prefer_node` 可以用来选择一个特定的节点, 只是当特定节点不可用时转而使用其他节点。
- ❑ `_shards:0,1` : 这个 `preference` 参数值让我们指定操作在哪个分片上执行 (在我们的例子里, 是指所有分片, 因为在 `mastering` 索引里我们只有分片 0 和 1)。这是唯一可以和其他选项值组合的 `preference` 参数值。例如, 为了在本地的 0 和 1 分片上执行查询, 我们用分号连接 0,1 和 `_local`, 最终 `preference` 参数看起来是就像这样: `0,1;_local`。允许我们在一个分片上执行查询对于调试非常有用。
- ❑ 自定义字符串: 把 `_preference` 参数值设置为一个自定义字符串, 可以确保使用相同参数值的查询在相同的分片上执行。例如, 如果我们发送一个 `preference` 参数值为 `mastering_Elasticsearch` 的查询, 假定查询会在位于 node1 和 node2 节点的主分片上执行。之后如果我们发送另一个有同样 `preference` 参数值的查询, 那么第 2 个查询将在同样的分片上执行。这个功能在我们有不同的刷新频率, 并且不希望用户在重复查询时看到不同结果的时候可以帮助我们。

还遗漏了一件事, 就是 Elasticsearch 的默认行为。Elasticsearch 默认会在分片和副本之间随机执行操作。如果我们发送大量的请求, 那么最终每个分片和副本上将会执行相同 (或者几乎相同) 数量的查询。

5.5 小结

在本章中，我们学习了如何为 Elasticsearch 的部署选择正确的副本分片数，我们也了解了在查询和索引时路由和别名是如何起作用的。同时学习了分片分配机制是如何工作的，以及我们怎样配置它来满足我们的需求。最后，我们还了解了查询执行偏好能够带给我们的各种特性。

在下一章中，我们将通过提供不同的 similarity 模型来深入探讨如何调整 Apache Lucene 的打分机制。我们将使用解码器 (codecs) 来修改倒排索引的格式，还将探讨准实时索引和查询，强制写入 (flush) 和刷新 (refresh) 操作，以及如何配置事务日志。我们还将讨论 IO 节流和索引段合并的相关知识。最后，还将介绍 Elasticsearch 的各种缓存，包括字段数据缓存、过滤器缓存和查询分片缓存。

底层索引控制

在上一章中，我们介绍了一般的分配策略及索引本身的架构。最开始，读者了解了如何选择正确的分片数和副本数，如何在索引期及查询期结合别名使用路由功能。同时也讨论了如何调整索引分片的分配行为。最后，我们讨论了什么是查询偏好，以及它能为用户带来什么。

在本章中，将会深入探讨 Elasticsearch 如何在底层处理索引分片的。到本章结束，将涵盖以下内容：

- ☐ 使用多种相似度模型来改变 Apache Lucene 评分
- ☐ 使用编解码器改变索引写入方式
- ☐ 准实时索引及搜索
- ☐ 数据提交，索引更新及事务日志处理
- ☐ I/O 节流
- ☐ 段合并控制及可视化
- ☐ Elasticsearch 缓存

6.1 改变 Apache Lucene 的评分方式

自 2012 年 Apache Lucene 4.0 发布以后，用户有机会改变默认的基于 TF/IDF 的评分算法。Lucene 的 API 做了一些改变，使得用户能轻松地修改和扩展该评分公式。不过，这并不是 Lucene 在改变文档评分计算方面仅有的改进。Lucene 4.0 提供了更多的相似度模型，

这允许我们采用不同的评分公式。本节中，我们将深入了解 Lucene 4.0 带来了哪些变化，以及如何整合这些特性至 Elasticsearch 中。

6.1.1 可用的相似度模型

前面已经提到过了，Apache Lucene 4.0 之前，除了最原始和默认的相似度模型以外，TF/IDF 模型也是可用的。我们已经在 2.1 节中详细讨论过了。

而现在，有以下 5 种新的相似度模型可用。

- ❑ **Okapi BM25 模型**：它是一种基于概率模型的相似度模型，可以用来估算文档与给定查询匹配的概率。为了在 Elasticsearch 中使用它，你需要使用该模型的名字，BM25。一般来说 Okapi BM25 模型在短文本文档上的效果最好，这种场景中重复词项对文档的总体得分损害较大。
- ❑ **随机偏离（divergence from randomness）模型**：这是一种基于同名概率模型的相似度模型。为了在 Elasticsearch 中使用它，你需要使用该模型的名字 DFR。一般来说，随机偏离模型在类似自然语言的文本上使用效果较好。
- ❑ **基于信息的（information based）模型**：该模型与随机偏离模型类似。为了在 Elasticsearch 中使用它，你需要使用该模型的名字，IB。与 DFR 模型类似，IB 模型在类似自然语言的文本上使用也有较好的效果。
- ❑ **LM Dirichlet 模型**：该相似度模型结合了狄利克雷先验与贝叶斯平滑。为了在 Elasticsearch 中使用它，你需要使用该模型的名字，LMDirichlet。更多细节请参考：https://lucene.apache.org/core/4_9_0/core/org/apache/lucene/search/similarities/LMDirichletSimilarity.html。
- ❑ **LM Jelinek Mercer 模型**：该相似度模型使用了 Jelinek Mercer 平滑方法。为了在 Elasticsearch 中使用它，你需要使用该模型的名字，LMJelinekMercer。更多细节请参考：https://lucene.apache.org/core/4_9_0/core/org/apache/lucene/search/similarities/LMJelinekMercerSimilarity.html。



前面提到的相似度模型所涉及的数学知识已经远超过本书的讨论范围。如果您想深入了解这些模型以及拓展相关知识，请参考 http://en.wikipedia.org/wiki/Okapi_BM25（Okapi BM25 模型），及 http://terrier.org/docs/v3.5/dfc_description.html（随机偏离模型）。

6.1.2 为每字段配置相似度模型

自 Elasticsearch 0.90 以后，允许用户在映射中为每字段设置不同的相似度模型。例如，假

设我们有下面这个映射，用于索引博客的回帖（该映射存储在 `posts_no_similarity.json` 文件中）：

```
{
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes" },
        "name" : { "type" : "string", "store" : "yes", "index" :
"analyzed" },
        "contents" : { "type" : "string", "store" : "no", "index" :
"analyzed" }
      }
    }
  }
}
```

我们希望的是，在 `name` 字段和 `contents` 字段中使用 BM25 相似度模型。为了实现这个目的，我们需要扩展我们的字段定义，添加 `similarity` 字段，并将该字段的值设置为相应的相似度模型的名字。修改后的映射（该映射存储在 `posts_similarity.json` 文件中）如下所示：

```
{
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes" },
        "name" : { "type" : "string", "store" : "yes", "index" :
"analyzed", "similarity" : "BM25" },
        "contents" : { "type" : "string", "store" : "no", "index" :
"analyzed", "similarity" : "BM25" }
      }
    }
  }
}
```

以上更改就足够了，并不需要额外的信息。经过前面的处理，Apache Lucene 将在搜索期在 `name` 字段和 `contents` 字段上使用 BM25 相似度模型计算文档得分。



注意 对于随机偏离模型及基于信息的模型，我们需要配置一些额外的属性，用于控制这些相似度模型的行为。后续小节将会覆盖相关知识。

6.1.3 相似度模型配置

我们已经知道如何为索引中每个字段配置相似度模型了，现在让我们了解如何按需求配置它们。事实上，这是相当容易的。我们所要做的就是，在索引配置相关部分提供相应的相似度模型配置信息，例如，就像下面的代码这样（本范例存储在 `posts_custom_similarity.json` 文件中）：

```

{
  "settings" : {
    "index" : {
      "similarity" : {
        "mastering_similarity" : {
          "type" : "default",
          "discount_overlaps" : false
        }
      }
    }
  },
  "mappings" : {
    "post" : {
      "properties" : {
        "id" : { "type" : "long", "store" : "yes" },
        "name" : { "type" : "string", "store" : "yes", "index" :
"analyzed", "similarity" : "mastering_similarity" },
        "contents" : { "type" : "string", "store" : "no", "index" :
"analyzed" }
      }
    }
  }
}

```

尽管用户可以配置多个相似度模型，但此时我们还是把目光投向前面的范例。我们定义了一个名叫 `mastering_similarity` 的新的相似度模型，它基于默认的 TF/IDF 相似度模型。并且将它的 `discount_overlaps` 属性值设置为 `false`，指定该相似度模型用于 `name` 字段。后面我们将讨论不同相似度模型都有哪些属性，现在，让我们关注一下如何改变 Elasticsearch 的默认相似度模型。

6.1.4 选择默认的相似度模型

为了设置默认的相似度模型，我们需要提供一个名为 `default` 的相似度模型的配置信息。例如，如果我们想使用 `mastering_similarity` 模型作为默认的相似度模型，我们需要将前面的配置文件修改为如下形式（该范例被存储在 `posts_default_similarity.json` 文件中）：

```

{
  "settings" : {
    "index" : {
      "similarity" : {
        "default" : {
          "type" : "default",
          "discount_overlaps" : false
        }
      }
    }
  },
  ...
}

```


由于所有的相似度模型都全局使用了 query norm 和 coord 这两个评分因子（详情见 2.1 节），但是它们在 default 相似度模型的配置中被移除了。Elasticsearch 允许用户根据需要改变这种状况。为了实现该目的，用户需要定义个另外一个名为 base 的相似度模型。它的定义方式与前面的范例如出一辙，将相似度模型的名字由 default 改为 base 即可。可参考下面的代码（该范例代码被保存在 posts_base_similarity.json 文件中）：

```
{
  "settings" : {
    "index" : {
      "similarity" : {
        "base" : {
          "type" : "default",
          "discount_overlaps" : false
        }
      }
    }
  },
  ...
}
```

如果 base 相似度模型出现在索引配置中，当 Elasticsearch 使用其他相似度模型计算文档得分时，则使用 base 相似度模型来计算 query norm 和 coord 评分因子。

配置被选用的相似度模型

每个新增的相似度模型都可以根据用户需求进行配置。Elasticsearch 允许用户不加配置而直接使用 default 和 BM25 相似度模型。这是因为它们是预先配置好的。而 DFR 和 IB 模型则需要进一步配置才能被使用。现在，让我们看看每个相似度模型都提供了哪些可配置的属性。

（1）配置 TF/IDF 相似度模型

在 TF/IDF 相似度模型案例中，我们可以只设置一个参数：discount_overlaps 属性，其默认值为 true。默认情况下，位置增量（position increment）为 0（即该词条的 position 计数与前一个词条相同）的词条在计算评分时并不会被考虑进去。如果在计算文档时需要考虑这类词条，则需要将相似度模型的 discount_overlaps 属性值设置为 false。

（2）配置 Okapi BM25 相似度模型

在 Okapi BM25 相似度模型案例中，有如下参数可以配置。

- k1：该参数为浮点数，控制饱和度（saturation），即词频归一化中的非线性项。
- b：该参数为浮点数，用于控制文档长度对词频的影响。
- discount_overlaps：与 TF/IDF 相似度模型中的 discount_overlaps 参数作用相同。

（3）配置 DFR 相似度模型

在 DFR 相似度模型案例中，有如下参数可以配置。

□ `basic_model`: 该参数值可设置为 `be`、`d`、`g`、`if`、`in`, 以及 `ine`。

□ `after_effect`: 该参数值可设置为 `no`、`b`, 以及 `l`。

□ `normalization`: 该参数值可设置为 `no`、`h1`、`h2`、`h3` 以及 `z`。

如果 `normalization` 参数值不是 `no`, 则需要设置归一化因子。归一化因子的设置依赖于我们选择的 `normalization` 参数值。参数值为 `h1` 时, 使用 `normalization.h1.c` 属性; 参数值为 `h2` 时, 使用 `normalization.h2.c` 属性; 参数值为 `h3` 时, 使用 `normalization.h3.c` 属性; 参数值为 `z` 时, 使用 `normalization.z.z` 属性。这些属性值的类型均为浮点数。下面的代码片段展示了如何配置相似度模型:

```
"similarity" : {
  "esserverbook_dfr_similarity" : {
    "type" : "DFR",
    "basic_model" : "g",
    "after_effect" : "l",
    "normalization" : "h2",
    "normalization.h2.c" : "2.0"
  }
}
```

(4) 配置 IB 相似度模型

在 IB 相似度模型案例中, 有如下参数可以配置。

□ `distribution`: 该参数值可设置为 `ll` 或 `spl`。

□ `lambda`: 该参数值可设置为 `df` 或 `tff`。

此外, IB 模型也需要配置归一化因子, 它的配置方式与 DFR 模型相同, 故不赘述。下面的代码片段展示了如何配置 IB 相似度模型:

```
"similarity" : {
  "esserverbook_ib_similarity" : {
    "type" : "IB",
    "distribution" : "ll",
    "lambda" : "df",
    "normalization" : "z",
    "normalization.z.z" : "0.25"
  }
}
```

(5) 配置 LM Dirichlet 相似度模型

在 LM Dirichlet 相似度模型案例中, 有如下参数可以配置。

□ `mu`: 该参数可配置, 默认值为 2000。

下面是该模型参数配置的例子:

```
"similarity" : {
  "esserverbook_lm_dirichlet_similarity" : {
```

```

    "type" : "LMDirichlet",
    "mu" : "1000"
  }
}

```

(6) 配置 LM Jelinek Mercer 相似度模型

在 LM Jelinek Mercer 相似度模型案例中，有如下参数可以配置。

□ **lambda**: 该参数可配置，默认值为 0.1。

下面是该模型参数配置的例子：

```

"similarity" : {
  "esserverbook_lm_jelinek_mercer_similarity" : {
    "type" : "LMJelinekMercer",
    "lambda" : "0.7"
  }
}

```



注意 一般来说，较短字段（例如文档的 title 字段），lambda 的值可设置在 0.1 左右，而较长字段，lambda 值应该设置为 0.7。

6.2 选择适当的目录实现——store 模块

当配置 Elasticsearch 集群时，有些模块往往容易被用户所忽略，store 模块正是其中之一。然而该模块非常重要，它是 Apache Lucene 与其 I/O 子系统之间的一个抽象。Lucene 所有在磁盘上的操作都通过它的 store 模块来处理。而 Elasticsearch 中绝大多数 store 类型都是与 Lucene 的 Directory 类是一一对应的（参考：http://lucene.apache.org/core/4_9_0/core/org/apache/lucene/store/Directory.html）。目录（Directory）用来读写所有 Lucene 索引相关的文件，因此它的配置的非常重要。

store 类型

Elasticsearch 提供了 5 种可用的 store 类型。现在我们来看看都有哪些 store 类型，以及如何利用它们的特性。

1. 简单文件系统 store 类型

Directory 类的最简单实现就是使用一个随机存取文件（Java 中的 `RandomAccessFile` - <http://docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.html>）。对应的 Apache Lucene 中的子类为 `SimpleFSDirectory`（http://lucene.apache.org/core/4_9_0/core/org/apache/

lucene/store/SimpleFSDirectory.html)。对于简单的应用，该 store 类型足够了。其瓶颈在于多线程读写，会导致很糟糕的性能。在 Elasticsearch 中，通常使用基于 NIO 的 store 类型来替换简单文件系统 store 类型。尽管如此，我们还是要介绍一下该 store 类型的配置实用，如果想使用简单文件系统 store 类型，设置 index.store.type 属性值为 simplefs。

2. NIO 文件系统 store 类型

该 store 类型使用了 Directory 类基于 java.nio 包中 FileChannel 类的实现（可参考 <http://docs.oracle.com/javase/7/docs/api/java/nio/channels/FileChannel.html>），该类型对应的是 Apache Lucene 中的 NIOFSDirectory 类。该类型允许许多线程操作同一个文件，同时不用担心性能急剧下降。为了使用该 store 类型，需要将 index.store.type 属性值设置为 niofs。



注意 读者请注意，由于针对 Windows 平台的 JVM 存在一些 bug，在 Windows 上使用 niofs 很可能存在性能问题。想了解更多此类 bug 的信息，可参考：http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6265734。

3. MMap 文件系统 store 类型

该 store 类型使用了 Apache Lucene 中的 MMapDirectory 类（详情可参考：http://lucene.apache.org/core/4_9_0/core/org/apache/lucene/store/MMapDirectory.html）。它使用了 mmap 系统调用（详情可参考：<http://en.wikipedia.org/wiki/Mmap>）处理读操作，使用随机读写文件处理写操作。读文件时，会将文件映射到同样的大小的虚拟地址空间中去（如果有足够的空间的话）。因为 mmap 没有加锁操作，因此在多线程读写的时候是可扩展的。当使用 mmap 读取索引文件时，感觉像是事前缓存了该文件一样（因为它被映射到虚拟地址空间中去了）。由于这个原因，此时读取 Apache Lucene 的索引文件，并不需要加载文件到操作系统缓存中去，因此访问会更快。该 store 类型等价于允许 Lucene 或 Elasticsearch 直接访问 I/O 缓存，因此读写索引文件速度更快。

在 64 位操作系统中使用 MMap 文件系统 store 类型毫无意义，应仅在 32 位操作系统中使用它，并确保索引足够小并且有足够的虚拟地址空间。如果想使用 MMap 文件系统 store 类型，应将 index.store.type 属性值设置为 mmap。

4. 混合文件系统 store 类型

该 store 类型出现在 Elasticsearch 1.3.0 中，它根据具体的文件类型，混合使用 NIO 及 MMap。在本书撰写之时，只针对词典文件及 doc values 文件使用 MMap，而其他索引文件读写使用 NIO。如果想使用混合文件系统 store 类型，应将 index.store.type 值设置为 default。

5. 内存 store 类型

这是第 2 种不基于 Apache Lucene Directory 的 store 类型（第 1 种是混合文件系统 store 类型）。该类型允许直接将索引文件存储在内存中，而不是磁盘中。这个事实很重要，因为文件是存储在内存中的，因此不可持久化。如果碰上集群重启这样的事情，内存中的索引文件将会丢失。但是，如果用户想构建小容量、高性能，同时又有多个分片或副本以及能快速重建的索引，该 store 类型就非常值得期待了。如果想使用内存 store 类型，应将 `index.store.type` 值设置为 `memory`。



注意 内存 store 类型与其他 store 类型类似，其存储的数据能在多个节点之间被复制。

额外的属性

当使用内存 store 类型，用户可以进行某种程度的控制。允许控制内存 store 特性，这很重要。下面是可配置的内存 store 参数（节点级控制）。

- ❑ `cache.memory.direct`：默认值为 `true`，该参数用于确定是否在 JVM 堆内存之外分配内存。一般来说该参数保持为默认值是非常合适的，因为这样会避免 JVM 内存过度使用。
- ❑ `cache.memory.small_buffer_size`：默认值为 1kB，该参数指定了小块缓冲区的大小，这里的缓冲区指的是一个内部内存结构，用来保存段信息及被删除文档的信息。
- ❑ `cache.memory.large_buffer_size`：默认值为 1MB，该参数指定了大块缓冲区的大小，这里的缓冲区指的是一个内部内存结构，用来保存除段信息及被删除文档信息之外的索引文件。
- ❑ `cache.memory.small_cache_size`：数据对象的小块缓存的大小，这是一个内部的内存结构，用来缓存段信息及被删除文件信息。默认值为 10MB。
- ❑ `cache.memory.large_cache_size`：数据对象的大块缓存的大小，这是一个内部的内存结构，用来缓存段信息及被删除文件信息以外的索引信息。默认值为 500MB。

6. 默认 store 类型

Elasticsearch 1.3.0 与之前及之后的版本中，默认 store 类型有所不同。

7. Elasticsearch 1.3.0 及以后版本默认 store 类型

从 Elasticsearch 1.3.0 开始，默认的 store 类型为混合文件系统类型，可以通过设置 `index.store.type` 值为 `default` 来实现。

8. Elasticsearch 1.3.0 之前版本默认 store 类型

从 Elasticsearch 1.3.0 往前，默认 store 类型是基于文件系统的，但是具体选项会因操作系统而异。例如，32 位的 Windows 操作系统中，默认使用 `simplefs`，而在 Solaris 或 64 位 Windows 操作系统中则使用 `mmap`，其余的操作系统默认使用 `niofs` 类型。

**注意**

如果想深入了解 Directory 的底层实现, 可参考 Uwe Schindler 的博客: <http://blog.thetaphi.de/2012/07/use-lucenes-mmapdirectoryon-64bit.html>, 及 Jörg Prante 的博客: <http://jprante.github.io/lessons/2012/07/26/Mmap-withLucene.html>。

通常来说, 默认 store 类型就是您想要的。有时候也会有所不同, 例如也许内存量足够大, 索引也非常大, 此时使用 mmap 类型会更合适。当使用 mmap 读取索引文件时, 会一次性缓存索引数据, 然后 Apache Lucene 及操作系统能重复使用这些数据。

6.3 准实时、提交、更新及事务日志

一个理想的搜索解决方案中, 新索引的数据应该能立即被搜索到。Elasticsearch 给人的第一印象仿佛就是如此工作的, 即使是在多服务器环境下, 而事实上并不如此 (至少不是每种场景都能保证新索引的数据能被实时检索到), 后面我们将讲解原因。

接下来的案例中, 我们将使用下面的命令, 将一篇文档索引到新创建的索引中:

```
curl -XPOST localhost:9200/test/test/1 -d '{ "title": "test" }'
```

现在, 我们将更新该文档, 并尝试立即搜索它。为实现该目的, 我们将串行执行下面的两个命令:

```
curl -XPOST localhost:9200/test/test/1 -d '{ "title": "test2" }' ;
curl -XGET 'localhost:9200/test/test/_search?pretty'
```

前面的命令将返回类似下面的结果:

```
{ "_index": "test", "_type": "test", "_id": "1", "_version": 2, "created": false },
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "test",
      "_type" : "test",
      "_id" : "1",
      "_score" : 1.0,
      "_source": { "title": "test" }
    } ]
  }
}
```

我们把两个结果放在一起考察。第 1 个返回结果对应索引命令的操作。正如你所见，一切正常（成功更新了文档，可查看返回结果的 `_version` 字段）。第 2 个返回结果是查询对应的结果，它的 `title` 字段的值应该是 `test2`，然而返回的是修改前的那个文档，这是怎么回事？关于上面这个问题，在给出答案之前，不妨去回顾一下 Lucene 的内部机制，探究一下 Lucene 是如何让新索引的文档在搜索时可用。

6.3.1 索引更新及更新提交

在阅读 1.1 节时我们已经知道，索引期新文档被写入索引段。索引段是独立的 Lucene 索引，这意味着查询是可以与索引并行进行的，只是不时有新增的索引段被添加至可被搜索的索引段集合之中。Apache Lucene 通过创建后续的（基于索引只写一次的特性）`segments_N` 文件来实现此功能，该文件列举了索引中的索引段。这个过程被称为提交（committing），Lucene 以一种安全的方式来执行该操作，能确保索引更改以原子操作方式写入索引。即便有错误发生，也能保证索引数据的一致性。

让我们回到之前的例子，尽管第一个操作添加了文档至索引中，但是它并没有执行提交操作。这就是返回结果令人惊讶的原因。然而，一次提交并不足以保证新索引的数据能被搜索到。这是因为 Lucene 使用一个叫作 `Searcher` 的抽象类来执行索引的读取。该类需要被刷新。

如果索引更新提交了，但是 `Searcher` 实例并没有重新打开，那么它觉察不到新索引段的加入。`Searcher` 重新打开的过程叫作刷新（refresh）。出于性能考虑，Lucene 推迟了耗时的刷新，因此它不会在每次新增一个文档（或每次批量增加文档）的时候刷新，但是 `Searcher` 会每秒钟刷新一次。这种刷新已经非常频繁了，然而有很多应用需要更快的刷新频率。如果碰到这种状况，要么使用其他技术，要么审视需求是否合理。Elasticsearch 提供了强制刷新的 API。例如，在我们的例子中，可以使用下面的命令：

```
curl -XGET localhost:9200/test/_refresh
```

如果我们在搜索之前执行了该命令，那么将会得到我们预期的结果。

更改默认的刷新时间

`Searcher` 自动刷新的时间间隔可以通过以下手段改变：通过更改 Elasticsearch 配置文件中的 `index.refresh_interval` 参数值或者使用配置更新相关的 API。例如：

```
curl -XPUT localhost:9200/test/_settings -d '{
  "index" : {
    "refresh_interval" : "5m"
  }
}'
```


上面的命令将 Searcher 的自动刷新时间间隔更改为 5 分钟。请注意，上次刷新之后新增的数据并不会被搜索到。



注意

刷新操作是很耗资源的，因此刷新间隔时间越长，索引速度越快。如果您需要长时间高速建索引，并且在建索引结束之前并不执行查询。那么可以考虑将 `index.refresh_interval` 参数设置为 -1，然后在建索引结束以后将该参数恢复为初始值。

6.3.2 事务日志

Apache Lucene 能保证索引的一致性，这非常棒。但是这并不能保证当我们往索引中写数据失败时不损失数据（例如，磁盘空间不足、设备损坏，或没有足够的文件句柄供索引文件使用）。另外一个问题是频繁提交会导致严重的性能问题（因为每次提交会触发一个索引段的创建操作，同时也可能触发索引段的合并）。Elasticsearch 通过使用事务日志（transaction log）来解决这些问题。事务日志用来保存所有的未提交的事务，Elasticsearch 会不时创建一个新的日志文件用于记录每个事务的后续操作。当有错误发生时，事务日志将会被检查，必要时会再次执行某些操作，以确保没有丢失任何更改。事务日志的相关操作都是自动完成的，用户并不会意识到某个特定时刻触发的更新提交。事务日志中的信息与存储介质之间的同步（同时清空事务日志）被称为事务日志刷新（flushing）。



注意

请注意事务日志刷新与 Searcher 刷新的区别。大多数情况下，Searcher 刷新是你所期望的，即搜索到最新的文档。而事务日志刷新用来保障数据正确写入了索引并清空事务日志。

除了自动的事务日志刷新以外，也可以使用对应的 API。例如，我们可以使用下面的命令，强制将事务日志中涉及的所有的数据更改操作同步到索引中，并清空事务日志文件：

```
curl -XGET localhost:9200/_flush
```

我们也可以使用 `flush` 命令对特定的索引进行事务日志刷新（比如说我们的索引名为 `library`）：

```
curl -XGET localhost:9200/library/_flush
```

```
curl -XGET localhost:9200/library/_refresh
```

上面第 2 行命令中，我们紧接着在事务日志刷新之后，调用了 Searcher 刷新操作，打开了一个新的 Searcher 实例。

事务日志相关配置

如果事务日志的默认配置不能满足用户需要，Elasticsearch 允许用户修改默认配置以满

足特定需求。以下参数可以通过修改 Elasticsearch.yml 文件来配置，也可以通过索引配置更新 API 更改配置。

- ❑ `index.translog.flush_threshold_period`：该参数默认值为 30 分钟，它控制了强制自动事务日志刷新的时间间隔，即便是没有新数据写入。强制进行事务日志刷新通常会导致大量的 I/O 操作，因此有时当事务日志涉及少量数据时，更适合进行频繁的事务日志刷新操作。
- ❑ `index.translog.flush_threshold_ops`：该参数确定了一个最大的操作数，在上次事务日志刷新以后，当索引更改操作次数超过该数，则强制进行事务日志刷新操作，默认情况下不限制操作次数。
- ❑ `index.translog.flush_threshold_size`：该参数确定了事务日志最大容量，当容量超过该参数值，就强制进行事务日志刷新操作，默认值为 200MB。
- ❑ `index.translog.interval`：该参数默认值为 5s，它描述了连续两次事务日志刷新检查之间的周期。Elasticsearch 会将该值随机赋值为预设值及其预设值 x2 之间的一个数。
- ❑ `index.gateway.local.sync`：该参数定义了通过 fsync 系统调用同步事务日志数据的频率，默认情况是 5s 一次。
- ❑ `index.translog.disable_flush`：禁用事务日志刷新。尽管默认情况下事务日志刷新是可用的，但是有时候临时性禁用它能带来其他方面的便利。例如，向索引中导入大量文档的时候。



注意 尽管前面提及的所有参数都被指定用于用户选定的某个索引，但它们同时也定义了该索引的每个分片的事务日志处理方式。

当然，除了通过修改 Elasticsearch.yml 文件来配置上述参数，我们也可以使用 API 更改相关配置。例如：

```
curl -XPUT localhost:9200/test/_settings -d '{
  "index" : {
    "translog.disable_flush" : true
  }
}'
```

如果在向索引导入大量文档之前执行上述操作，则会大幅度提高索引的速度。但是请记住，当导入数据完毕之后，要重新设置事务日志刷新相关参数。

6.3.3 准实时读取

事务日志给我们带来一个免费的特性：实时读取（real-time GET），该功能提供了返回

文档各种版本（包括未提交版本）的可能性。实时读取操作从索引中读取数据时，它会先检查事务日志中是否有可用的新版本。如果近期索引没有与事务日志同步，那么索引中数据将会被忽略，事务日志中的最新版本的文档将会被返回。

为了演示实时读取的工作原理，我们用下面的命令替换范例中的搜索操作：

```
curl -XGET localhost:9200/test/test/1?pretty
```

Elasticsearch 将会返回类似下面的结果：

```
{
  "_index" : "test",
  "_type" : "test",
  "_id" : "1",
  "_version" : 2,
  "exists" : true, "_source" : { "title": "test2" }
}
```

如果你查看了该结果，你将会看到，这正是我们想要的结果，这里我们并没有使用刷新技巧就得到了最新版本的文档。

6.4 控制索引合并

读者已经知道（我们已经在第 1 章中讨论过），在 Elasticsearch 中每个索引都会创建一到多个分片以及 0 到多个副本。您当然也知道这些分片或副本本质上都是 Lucene 索引，而 Lucene 索引又基于多个索引段构建（至少一个索引段）。索引文件中绝大部分数据只写一次，读多次。索引中只有用于保存文档删除信息的文件会被多次更改。在某些时刻，当某种条件满足时，多个索引段会被拷贝合并到一个更大的索引段，而那些旧的索引段会被抛弃并从磁盘删除。这个操作被称为段合并（segment merging）。

也许你会有疑问，为什么非要进行段合并？有以下这些理由：首先，索引段的个数越多，搜索性能越低并且要耗费更多的内存。另外，索引段是不可变的，你并不能物理上从中删除信息。也许你碰巧从索引中删除了大量文档，这些文档只是做了删除标记，物理上并没有被删除。当段合并发生时，这些被标记为删除的文档并没有被拷贝至新的索引段中。这样的话，减少了最终索引段中的文档数。



注意

频繁的文档更改会导致大量的小索引段。这会导致文件句柄打开过多的问题。我们必须要对这种情况有所准备，比如说，修改系统配置，设置合适的最大文件打开数。

从用户角度来看，可快速地在两个方面对段合并做如下概括：

□ 当一些索引段合并为一个索引段的时候，会减少索引段的数量并提高搜索速度。

- 同时也会减少索引的容量（文档数），因为在段合并时会移除被标记为已删除的那些文档。

尽管段合并有这些好处，但是用户也应该了解到段合并也有它的代价。主要是 I/O 操作的代价。在速度较慢的系统，段合并会显著影响性能。基于这个原因，Elasticsearch 允许用户选择段合并策略（merge policy）及存储级节流（store level throttling）。

6.4.1 选择正确的合并策略

尽管段合并是 Lucene 的责任，Elasticsearch 也允许用户配置想用的段合并策略。到目前为止，有 3 种可用的合并策略：

- tiered（默认选项）
- log_byte_size
- log_doc

前面提到的每一种段合并策略都有它们自己的参数，这些参数定义了各自的行为特点，并且这些参数的默认值都是可以被改写的（请阅读后续章节来了解这些参数）。

为了告诉 Elasticsearch 我们想使用哪个段合并策略，可以将配置文件的 `index.merge.policy.type` 字段配置成我们期望的段合并策略类型。例如像下面这样：

```
index.merge.policy.type: tiered
```



注意 一旦使用特定的段合并策略创建了索引，它将不能被改变。但是，可以使用相关 API 改变该段合并策略的参数值。

接下来我们可以了解这些不同的段合并策略，以及它们提供的功能。此后我们将讨论这些段合并策略的具体配置。

1. tiered 合并策略

这是 Elasticsearch 的默认选项。它合并大小相似的索引段，并考虑了每层允许的索引段的最大个数。读者需要区分单次可合并的索引段的个数与每层允许的索引段数的区别。在索引期间，该合并策略将会计算索引中允许出现多少个索引段，该数值被称为阈值（budget）。如果正在构建的索引中的段数超过了阈值，该策略将先对索引段按容量降序排序（这里考虑了被标记为已删除的文档），然后选择一个成本最低的合并。合并成本的计算方法倾向于回收更多删除文档及产生更小的索引段。

如果某次合并产生的索引段的大小大于 `index.merge.policy.max_merged_segment` 参数值，则该合并策略将会选择更少的索引段参与合并，使得生成的索引段的大小小于阈值。这意味着，对于有多个分片的索引，默认的 `index.merge.policy.max_merged_segment` 则显

得过小，会导致产生大量的索引段的创建，从而降低了查询速度。用户应该根据自己具体的数据量，观察索引段的状况，不断调整合并策略以满足应用需求。

2. log byte size 合并策略

该合并策略将会不断地以字节数的对数为计算单位，选择多个索引合并创建新索引。合并过程中，某个时刻会有一些较大的索引段，然后又会产生少于合并因子（merge factor）的一些索引段，如此循环往复。你可以想象，时而有一些相同数量级的索引段，然后索引段的个数变得比合并因子还少。当碰到一个特别大的索引段时，所有小于该级别的索引段将被合并。索引中的索引段个数与下次用于计算的字节数的对数成正比。该合并策略能够保持较少的索引段数量并且极小化段索引合并的代价。

3. log doc 合并策略

该策略与 log_byte_size 合并策略类似，不同的是前者基于索引的字节数计算，而后者基于索引段的文档数计算。以下两种情况中该合并策略表现良好：文档集中文档大小类似，或者你期望参与合并的索引段在文档数方面相似。

6.4.2 合并策略配置

我们现在已经知道索引的段合并策略的工作原理了，但是还缺乏配置方面的相关知识。所以现在可以讨论一下每种合并策略及其提供的配置选项。请记住，大多数情况下默认选项是够用的，除非有特殊的需求才需要修改。

1. 配置 tiered 合并策略

当使用 tiered 合并策略时，以下这些选项可配置。

- ❑ index.merge.policy.expunge_deletes_allowed：默认值为 10，该值用于确定被删除文档的百分比，当执行 expungeDeletes 时，该参数值用于确定索引段是否被合并。
- ❑ index.merge.policy.floor_segment：该参数用于阻止频繁刷新微小索引段。小于该数值的索引段由索引合并机制处理，此时视这些索引段的大小为该参数值。默认值为 2MB。
- ❑ index.merge.policy.max_merge_at_once：该参数确定了索引期单次合并涉及的索引段数量的上限。默认值为 10。该参数值较大时，允许更多的索引段参与单次合并，但是会消耗更多的 I/O 资源。
- ❑ index.merge.policy.max_merge_at_once_explicit：该参数确定了索引优化（optimize）操作及 expungeDeletes 操作能参与的索引段数量的上限。默认值为 30。该值对索引期参与合并的索引段数量的上限没有影响。

- ❑ `index.merge.policy.max_merged_segment`：该参数默认值为 5GB，它确定了索引期段合并中产生的单个索引段大小的上限。这是一个近似值，只是因为合并后产生的索引段的大小是通过累加参与合并的索引段的大小再减去被删除文档的大小而来的。
- ❑ `index.merge.policy.segments_per_tier`：该参数确定了每层允许出现的索引段数量的上限。该参数值越小，则导致更少的索引段数量，这也意味着更多的合并操作以及更低的索引性能。默认值为 10，可以设置为大于等于 `index.merge.policy.max_merge_at_once`，否则你将遇到很多与索引合并以及性能相关的问题。
- ❑ `index.reclaim_deletes_weight`：该参数值默认为 2.0，它确定了索引合并操作中清除被删除文档这个因素的权重。如果该参数设置为 0，则清除被删除文档对索引合并没有影响。该值越高，则清除更多被删除文档的合并更受合并策略青睐。
- ❑ `index.compound_format`：该参数值类型为布尔值，它确定了索引是否存储为复合文件格式（compound format）。默认值为 false。如果设置为 true，则 Lucene 将所有文件存储在一个文件中。有时候这样设置能解决操作系统打开文件句柄过多的问题，但是也会降低索引和搜索的性能。

2. 配置 log byte size 合并策略

当采用 `log_byte_size` 合并策略时，以下选项可配置。

- ❑ `merge_factor`：该参数确定了索引期间索引段以多快的频率进行合并。该值越小，搜索的速度越快，消耗的内存越少，而其代价则是更慢的索引速度。如果该值设大，情形则正好相反，更快的索引速度（因为索引合并更少），搜索速度更慢，消耗的内存更多。该参数默认值为 10.0，对于批量索引构建，可以设置较大的值，对于日常索引维护则可采用较小的值。
- ❑ `min_merge_size`：该参数定义了索引段可能的最小容量（段中所有文件的字节数）。如果索引段大小小于该参数值，并且 `merge_factor` 参数值允许，则进行索引段合并。默认值为 1.6MB，该参数对于避免产生大量小索引段是非常有用的。然而，用户应该记住，该参数值设置为较大值时，将会导致较高的合并成本。
- ❑ `max_merge_size`：该参数定义了允许参与合并的索引段的最大容量（以字节为单位），默认情况下不设置。因此默认情况下，在索引合并时对索引段大小没有限制。
- ❑ `maxMergeDocs`：该参数定义了参与合并的索引段的最大文档数。默认情况下不设置，因此默认情况下索引合并时，对索引段没有最大文档数的限制。
- ❑ `calibrate_size_by_deletes`：该参数为布尔值，如果设置为 true，则段中被删除文档的数量用于索引段大小的计算。

3. 配置 log doc 合并策略

当使用 log_doc（文档数对数）合并策略时，可配置以下这些选项。

- ❑ merge_factor：与 log_byte_size 合并策略中该参数的作用相同，请参考前面的解释。
- ❑ min_merge_docs：该参数定义了最小索引段允许的最小文档数。如果某个索引段中文档数低于该参数值，并且 merge_factor 参数也允许合并，则将会执行索引合并。该参数默认值为 1000，该参数对于避免产生大量小索引段是非常有用的。但是用户需要记住，将该参数值设置过大将会增大索引合并的代价。
- ❑ max_merge_docs：该参数定义了可以参与索引合并的索引段的最大文档数。默认情况下不设置，因此默认情况下，对参与索引合并的索引段的最大文档数没有限制。
- ❑ calibrate_size_by_deletes：该参数为布尔值，如果设置为 true，则段中被删除文档的数量用于索引段大小的计算。

与前面介绍的合并策略类似，上面提及的属性需要以 index.merge.policy 为前缀。例如，如果我们想设置 min_merge_docs 属性，则应该设置 index.merge.policy.min_merge_docs 属性。

6.4.3 调度

除了可以影响索引合并策略的行为之外，Elasticsearch 还允许我们定制合并策略的执行方式。有两种索引合并调度器（scheduler），默认的是并发合并调度器 ConcurrentMergeScheduler。

1. 并发合并调度器

该调度器使用多线程执行索引合并操作。每次开启一个新线程直到线程数达到上限。如果达到线程数上限，而又必须开启新线程（因为需要进行新的索引合并），那么所有的索引操作将被挂起，直到至少一个索引合并操作完成。

为了控制最大线程数，可以通过修改 index.merge.scheduler.max_thread_count 属性来达到目的。一般来说，可以按如下公式来计算允许的最大线程数：

```
maximum_value(1, minimum_value(3, available_processors / 2))
```

如果我们的系统是 8 核的，那么调度器允许的最大线程数可以设置为 4。

读者请记住，当您使用机械硬盘时，该合并调度策略不是很合适。您很快产生这种念头：并发合并与磁盘的吞吐能力是否协调。一旦用户观察到很慢的合并，则应调小线程数量。一般来说，如果用户使用的是机械硬盘，多并发合并调度器的线程数应设置为 1。

2. 顺序合并调度器

该调度器非常简单，使用同一个线程执行所有的索引合并操作。它在执行合并时将会

导致该线程的其他文档处理都被挂起，这意味着索引操作会被推迟。该合并调度器的存在只是出于兼容性的考虑，事实上并发合并调度器将线程数设置为 1 时跟它是等价的。

3. 设置合并调度

为了设置想要的索引合并调度器，用户可设置 `index.merge.scheduler.type` 属性值设置为 `concurrent` 或 `serial`。例如，为了使用并发合并调度器，用户应该如此设置：

```
index.merge.scheduler.type: concurrent
```

如果想使用顺序合并调度器，用户则应该像下面这样设置：

```
index.merge.scheduler.type: serial
```



注意 当谈论到索引合并策略和调度器，可视化演示它是最好不过了。如果您想了解在 Lucene 之中索引合并是如何执行的，不妨参阅 Mike McCandless 的博客：<http://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html>。除此之外，也可以使用一个叫作 `SegmentSpy` 的插件。可参考下面这个 URL 中内容：<https://github.com/polyfractal/Elasticsearch-segmentspy>。

6.5 关于I/O调节

在本章前面的 6.2 节中，讨论了 `store` 类型，并有能力配置适合需求的 `store` 模块。不过我们并没有深入讨论 `store` 模块的方方面面，甚至没有提及 I/O 调节 (I/O throttling)。

6.5.1 控制 I/O 节流

在 6.4 节中我们了解到，Apache Lucene 把索引数据保存在允许一次写入多次读取的不可变索引段中。索引合并的过程是异步的，从 Lucene 的角度看是不会干扰索引和查询过程的。然而，很可能会出现这个问题，因为合并操作非常消耗 I/O，需要读取旧索引段然后合并写入新索引段中。如果在此同时进行查询和索引，I/O 子系统的负荷会非常大，这个问题对于那些 I/O 速度较慢的系统尤其突出。这就是 I/O 节流的切入点。我们可以控制 Elasticsearch 使用的 I/O 量。

6.5.2 配置

在节点级和索引级都可以配置 I/O 节流。这意味着你可以分别配置节点的资源使用量和索引的资源使用量。



1. 节流类型

在节点级配置节流，可以使用 `indices.store.throttle.type` 属性。它支持 `none`、`merge`、`all` 这 3 个属性值。`none` 为默认值，表示不做任何限制。`merge` 表示在节点上进行索引合并时限制 I/O 使用量。`all` 表示对所有基于 `store` 模块的操作都做 I/O 限制。

在索引级配置节流，可以使用 `index.store.throttle.type` 属性。它除了支持 `indices.store.throttle.type` 的所有属性值以外，还支持一个 `node` 属性值。`node` 表示使用节点级配置取代索引级配置。`node` 是默认值。

2. 每秒最大吞吐量

在上面两种配置中，无论使用索引级还是节点级的节流配置，我们都可以设置 I/O 可用的每秒最大字节数。取值可以设置为 10MB、500MB 或任意我们需要的值。如果是索引级的配置，可以使用 `index.store.throttle.max_bytes_per_sec` 属性；而如果是节点级的配置，可以使用 `indices.store.throttle.max_bytes_per_sec` 属性。



注意 以上配置都可以通过 `Elasticsearch.yml` 文件配置，也可以动态更新：使用集群更新设置接口来更新节点级配置，使用索引更新设置接口来更新索引级配置。

3. 节点的默认节流配置

在节点级，I/O 节流从 Elasticsearch 0.90.1 版本起就默认开启。`indices.store.throttle.type` 属性设置为 `merge`，`indices.store.throttle.max_bytes_per_sec` 属性设置为 20MB。而 0.90.1 版之前的 Elasticsearch 没有默认开启 I/O 节流。

4. 性能考虑

如果您使用的是 SSD（固态硬盘）或者查询对系统运行速度影响不大（例如索引期间不执行查询），可以考虑将节流功能彻底关闭。通过将 `indices.store.throttle.type` 属性值设置为 `none` 来实现该目的。该操作会导致 Elasticsearch 不使用任何存储级节流，转而对所有存储操作使用磁盘。

5. 配置示例

现在假定我们有一个 4 个节点的集群，我们需要给整个集群配置 I/O 节流。我们希望单节点的索引合并操作每秒最多处理 50MB 的数据。因为我们知道在这个限制下不会影响查询性能，而这正是我们的目的。为此，我们需要执行如下命令：

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "persistent": {
    "indices.store.throttle.type": "merge",
```




```
"indices.store.throttle.max_bytes_per_sec" : "50mb"
}
}'
```

此外，我们还有一个叫 `payments` 的索引。这个索引极少使用，而我们把它放在整个集群中最小的那台机器上。该索引是单分片的且没有副本。我们期望限制它的索引合并操作，只允许它每秒最多处理 10MB 的数据。因此在上一个命令的基础上，我们还需要执行如下命令：

```
curl -XPUT 'localhost:9200/payments/_settings' -d '{
  "index.store.throttle.type" : "merge",
  "index.store.throttle.max_bytes_per_sec" : "10mb"
}'
```

在执行完以上命令后，我们可以通过执行如下命令来检查索引设置：

```
curl -XGET 'localhost:9200/payments/_settings?pretty'
```

我们将得到如下 JSON 响应：

```
{
  "payments" : {
    "settings" : {
      "index" : {
        "creation_date" : "1414072648520",
        "store" : {
          "throttle" : {
            "type" : "merge",
            "max_bytes_per_sec" : "10mb"
          }
        },
        "number_of_shards" : "5",
        "number_of_replicas" : "1",
        "version" : {
          "created" : "1040001"
        },
        "uuid" : "M3lePTOvSN2jnDz1J0t4Uw"
      }
    }
  }
}
```

如你所见，通过更新索引设置、关闭再重新打开索引，我们最终让配置变更生效了。

6.6 理解Elasticsearch缓存

缓存在 Elasticsearch 里扮演着重要角色，尽管很多用户意识不到它的存在。它允许我们



在内存中存储之前使用过的数据并根据需要适时重用它们。当然，我们不可能缓存所有的数据，因为数据容量总是大于内存容量，另外内存的构建代价也非常高昂。在本节里，我们将了解 Elasticsearch 提供的各种缓存功能，以及如何控制使用这些功能。了解缓存是如何工作的对于理解 Elasticsearch 内部工作机制是非常有帮助的。

6.6.1 过滤器缓存

过滤器缓存是 Elasticsearch 中最简单的缓存。过滤器缓存是负责缓存查询中使用的过滤器的执行结果的。在第 2 章中，已经介绍过过滤器的原理及使用方法了。为举例说明，我们看看下面的这个查询：

```
{
  "query" : {
    "filtered" : {
      "query" : {
        "match_all" : {}
      },
      "filter" : {
        "term" : {
          "category" : "romance"
        }
      }
    }
  }
}
```

它会返回所有在 category 字段中包含 romance 词项的文档。可以看到，我们使用了 match_all 查询连同一个过滤器。现在，在这个查询第一次执行以后，每个有与该查询相同过滤器的查询都会重用其结果，这样节省了宝贵的 I/O 和 CPU 资源。

1. 过滤器缓存的种类

在 Elasticsearch 中有两种类型的过滤器缓存：索引级的和节点级的。因此我们可以选择配置索引级或节点级（默认选项）的过滤器缓存。因为我们不能总是预知给定索引会分配到哪里（实际上是指索引的分片和副本），于是无法预测内存的使用，所以不建议使用索引级的过滤器缓存。

2. 节点级的过滤器缓存配置

节点级过滤器缓存是默认的缓存类型，它应用于分配到给定节点上的所有分片（设置 index.cache.filter.type 属性为 node，或者不设置这个属性）。Elasticsearch 允许我们使用 indices.cache.filter.size 属性来配置这个缓存的大小。既可以使用百分数，例如 10%（默认值），也可以使用确定的数值，例如 1024MB。如果我们使用百分数，Elasticsearch 会按当



前节点的最大堆内存的百分比来计算内存使用量。

节点级过滤器缓存是 LRU 类型（最近最少使用）缓存，这意味着为了给新记录腾出空间，在删除缓存记录时，使用次数最少的那些会被删除。

3. 索引级过滤器缓存的配置

第 2 种过滤器缓存类型为索引级过滤器缓存。Elasticsearch 允许我们使用下面的属性来配置索引级过滤器缓存的行为。

- ❑ `index.cache.filter.type`：这个属性设置缓存的类型，我们可以使用 `resident`、`soft`、`weak` 或 `node`（默认值）。在 `resident` 缓存中的记录不能被 JVM 移除，除非我们想移除它们（通过使用 API，设置最大缓存大小，或者设置过期时间），并且也是因为这个原因而推荐使用它（填充过滤器缓存代价很高）。内存吃紧时，JVM 可以清除 `soft` 和 `weak` 类型的缓存，区别是在清理内存时，JVM 会先选择清除 `weak` 引用对象，然后才是 `soft` 引用对象。最后的 `node` 属性代表缓存将在节点级控制。
- ❑ `index.cache.filter.max_size`：这个属性指定能够被存储到缓存中的最大记录数（默认是 -1，代表无限制）。需要注意这个设置不是应用在整个索引上的，而是应用于指定索引的某个分片的某个索引段上的，所以内存的使用量会因索引的分片数和副本数以及索引中段数的不同而不同。通常来说，结合 `soft` 类型，使用默认的无限制的过滤器缓存就足够了。谨记，慎用某些查询以保证缓存的可重用性。
- ❑ `index.cache.filter.expire`：这个属性指定过滤器缓存中记录的过期时间，默认是 -1，代表永不过期。如果我们希望对过滤器缓存设置超时时长，我们可以设置最大空闲时间。例如，如果希望缓存在最后一次访问后再过 60 分钟过期，我们应当设置该属性值为 60m。



注意 如果你想阅读更多关于 Java `soft` 引用和 `weak` 引用的信息，请参考 Java 文档，尤其是关于这两个类型的说明：<http://docs.oracle.com/javase/8/docs/api/java/lang/ref/SoftReference.html> 和 <http://docs.oracle.com/javase/8/docs/api/java/lang/ref/WeakReference.html>。

6.6.2 字段数据缓存

字段数据缓存使用时机是当我们的查询涉及非倒排 (uninverted) 数据操作时。Elasticsearch 所做的是加载相关字段的全部数据到内存中，这样 Elasticsearch 就能够快速地基于文档访问这些值。这种缓存可以被 Elasticsearch 用于切面计算、聚合、脚本计算、基于字段值的排序等地方。当第一次执行非倒排数据相关操作时，Elasticsearch 会把所有相关字段数据加载入内存，默认情况下这些给定字段的数据不会被移除。因此，可以快速访问



索引文档中给定字段的值。需要注意的是，从硬件资源的角度来看，构建字段数据缓存代价通常很高，因为字段的所有数据都需要加载到内存中，这需要消耗 I/O 操作和 CPU 资源。



注意 对于每个我们用来排序或做切面计算的字段，其数据都需要被加载到内存中，所有的词项。这样做的代价非常高昂，尤其是应用于那些高基数的字段（拥有大量不同词项的字段）时。

1. 字段数据与文档值

Lucene 的 docvalues 及其在 Elasticsearch 中的实现，随着版本的演进变得越来越优异。从 Elasticsearch 1.4.0 开始，doc values 几乎与字段数据缓存一样快。因为 doc values 在索引期计算并与索引文件一起存储，所以它并不耗费很多内存。事实上，它只需要少量堆内存，而速度跟字段数据缓存非常接近。如果您在某些应用场景需要大量使用字段数据缓存，不妨考虑对此字段使用 doc values。仅需要添加 doc_values 属性，将其设置为 true，剩下的 Elasticsearch 会自行完成。



注意 在本书撰写之时，Elasticsearch 尚不支持在已分词的文本字段上使用 doc values，但是可以在其他类型的字段上使用。

例如，如果您想设置 year 字段使用 doc values，可按如下方式修改配置信息：

```
"year" : {
  "type" : "long",
  "ignore_malformed" : false,
  "index" : "analyzed",
  "doc_values" : true
}
```

如果您已经重建了索引，当执行那些需要 year 字段中非倒排数据的操作时，那么此时将会使用 doc values 而不是字段数据缓存。

2. 节点级字段数据缓存配置

在 Elasticsearch 0.90.0 里，如果我们没有修改配置的话，节点级字段数据缓存是默认的字段数据缓存类型，我们可以通过使用下面的属性来进行配置。

- ❑ `index.fielddata.cache.size`：这个属性指定了字段数据缓存的最大容量，既可以是一个百分比的值，例如 20%，也可以是一个绝对的内存大小，例如 10GB。如果我们使用百分数，Elasticsearch 会按当前节点的最大堆内存的百分比来计算内存使用量。字段数据缓存的大小默认没有限制，内存消耗应该被监控，因为缓存会消耗分配给 JVM



的大量内存。

- `index.fielddata.cache.expire`：这个属性指定字段数据缓存中记录的过期时间，默认被设为 `-1`，表示缓存中的记录永不过期。如果我们希望设置字段数据缓存过期时长，可以设置最大空闲时间。例如，如果希望缓存在最后一次访问后再过 60 分钟过期，我们应当设置属性值为 `60m`。请记住，字段数据缓存构建代价非常高昂，设置该参数时需要慎重考虑。



注意

如果想确保 Elasticsearch 使用节点级的字段数据缓存，应当设置 `index.fielddata.cache.type` 属性为 `node`，或者不设置这个属性。

3. 索引级字段数据缓存配置

与索引级过滤器缓存类似，我们也可以使用索引级别的字段数据缓存，但是因为同样的原因我们并不建议使用它。原因就是很难预测哪个分片或索引会分配到哪个节点上，因此我们没法预估缓存每个索引的字段数据缓存需要的内存量，而这会带来内存使用方面的问题（如当 Elasticsearch 进行 `rebalancing` 操作时）。

不过，如果你清楚你在做什么，并且清楚你想使用什么，那么你可以使用 `resident` 或者 `soft` 类型的字段数据缓存。这可以通过设置 `index.fielddata.cache.type` 属性为 `resident` 或 `soft` 来实现。跟我们在描述过滤器缓存时讨论过的情形类似，除非我们自己想删除，`resident` 类型的缓存是不能被 JVM 删除的，推荐在使用索引级字段数据缓存时使用 `resident` 类型的缓存。重建字段数据缓存代价很高，并且会影响 Elasticsearch 的查询性能。`soft` 类型的字段数据缓存在缺少内存时会被 JVM 清除掉。

4. 过滤

除了前面提到的配置选项外，Elasticsearch 还允许我们选择将哪些字段值加载到字段数据缓存中。这在某些情况下非常有用，尤其是在做基于字段数据排序或切面计算或聚合计算时。Elasticsearch 支持 3 种类型的字段数据过滤：基于词频，基于正则表达式，或基于两者的组合。

某些场景中，字段数据过滤非常有用，例如，你想从切面计算的结果中排除那些低频词项。有时候可能需要这样做：例如，我们知道索引中有些词项存在拼写错误，而这些词项一定是低基数词项。我们不想基于它们做切面计算，因此可以从数据里删除它们，在数据源里修正它们，或者使用过滤器从字段数据缓存中删除它们。这不仅在 Elasticsearch 的返回结果里排除了它们，同时因为更少的数据存储在内存中，还降低了字段数据缓存的总量。现在我们来了解一下可能的过滤选项。



(1) 添加字段数据过滤信息

为了引入字段数据缓存过滤信息，需要在映射文件的字段定义部分添加一个额外的对象：fielddata 对象及其子对象 filter。于是，扩展后的字段定义，以某个抽象的 tag 字段为例，看起来与下面的配置类似：

```
"tag" : {
  "type" : "string",
  "index" : "not_analyzed",
  "fielddata" : {
    "filter" : {
      ...
    }
  }
}
```

我们会在下一节里介绍 filter 对象里应该放些什么。

(2) 基于词频过滤

基于词频过滤允许我们只加载那些频率高于指定的最小值且低于指定的最大值的词项。词频最小值和最大值由 min 和 max 参数指定。词项的频率范围不是针对整个索引的，而是针对索引段的。同一个词项在段级和索引级的频率分布往往是不一样的，这个差别非常重要。参数 min 和 max 可以赋一个百分比值，例如 1% 是 0.01，50% 是 0.5，也可以赋一个绝对词频数。

除此之外，我们可以包含 min_segment_size 属性。这个属性指定了在构建字段数据缓存时，索引段应满足的最小文档数，小于该文档数的索引段不会被考虑。

例如，如果我们希望只保存来自容量不小于 100 的索引段，且词频在段中介于 1% 和 20% 之间的词项到字段数据缓存中，那么字段映射看起来会像是下面这样：

```
{
  "book" : {
    "properties" : {
      "tag" : {
        "type" : "string",
        "index" : "not_analyzed",
        "fielddata" : {
          "filter" : {
            "frequency" : {
              "min" : 0.01,
              "max" : 0.2,
              "min_segment_size" : 100
            }
          }
        }
      }
    }
  }
}
```

(3) 基于正则表达式过滤

除了可以基于词频过滤，也可以基于正则表达式过滤。这时只有匹配特定正则表达式的词项会被加载到字段数据缓存中。例如，如果我们希望只缓存来自 tag 字段的数据，也许是 Twitter 标签（以字符 # 开头），应这样配置映射：

```
{
  "book" : {
    "properties" : {
      "tag" : {
        "type" : "string",
        "index" : "not_analyzed",
        "fielddata" : {
          "filter" : {
            "regex" : "^#.*"
          }
        }
      }
    }
  }
}
```

(4) 基于正则表达式和词频过滤

可以组合前面讨论过的过滤方法。因此，如果我们想把 tag 字段的数据保存到字段数据缓存中，但是只缓存那些以字符 # 开头，且所在索引段至少有 100 个文档，并且词项在段中介于 %1 和 20% 之间的词项。那么我们可以做如下映射：

```
{
  "book" : {
    "properties" : {
      "tag" : {
        "type" : "string",
        "index" : "not_analyzed",
        "fielddata" : {
          "filter" : {
            "frequency" : {
              "min" : 0.1,
              "max" : 0.2,
              "min_segment_size" : 100
            },
            "regex" : "^#.*"
          }
        }
      }
    }
  }
}
```



注意

请注意，字段数据缓存不是在索引期间构建的，但却可以在查询期间重建，于是我们可以在运行时改变过滤行为，这可以通过使用映射 API 更新 `fielddata` 配置节来实现。然而，需谨记在改变字段数据缓存过滤设置后，应清空缓存。这可以通过使用清理缓存 API 来实现，可参考后面的 6.6.5 节。

(5) 一个过滤的例子

现在回到本小节刚开始的那个例子中。我们想排除切面计算结果中的低频词项。在我们的例子中，低频词项指的是词频最低的那 50% 的词项。当然，这个频率非常高，例子里只索引了 4 个文档，在生产环境中应该指定更低的值。为了验证过滤的效果，我们用下面的命令创建一个 books 索引：

```
curl -XPOST 'localhost:9200/books' -d '{
  "settings" : {
    "number_of_shards" : 1,
    "number_of_replicas" : 0
  },
  "mappings" : {
    "book" : {
      "properties" : {
        "tag" : {
          "type" : "string",
          "index" : "not_analyzed",
          "fielddata" : {
            "filter" : {
              "frequency" : {
                "min" : 0.5,
                "max" : 0.99
              }
            }
          }
        }
      }
    }
  }
}
```

然后我们使用 bulk API 索引一些文档（代码存储在随书的 `regex.json` 文件中）：

```
curl -s -XPOST 'localhost:9200/_bulk' --data-binary '
{ "index": { "_index": "books", "_type": "book", "_id": "1" } }
{ "tag": ["one"] }
{ "index": { "_index": "books", "_type": "book", "_id": "2" } }
{ "tag": ["one"] }
{ "index": { "_index": "books", "_type": "book", "_id": "3" } }
{ "tag": ["one"] }
{ "index": { "_index": "books", "_type": "book", "_id": "4" } }
```



```
{ "tag": ["four"] }
```

现在，运行下面的查询来验证一个简单的词项切面计算（前面已经讨论过了，切面计算及聚合计算使用到了字段数据缓存）：

```
curl -XGET 'localhost:9200/books/_search?pretty' -d ' {
  "query" : {
    "match_all" : {}
  },
  "aggregations" : {
    "tag" : {
      "terms" : {
        "field" : "tag"
      }
    }
  }
}'
```

查询响应如下：

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  .
  .
  .
  "aggregations" : {
    "tag" : {
      "doc_count_error_upper_bound" : 0,
      "sum_other_doc_count" : 0,
      "buckets" : [ {
        "key" : "one",
        "doc_count" : 3 } ]
    }
  }
}
```

就像你看到的那样，切面计算只涉及了词项“one”，其他4个被忽略了。如果我们假定词项four存在拼写错误，那么就已经达到目的。

5. 字段数据格式

字段数据缓存并不是一个简单的功能，因此它的实现需要尽可能地节省内存空间。由于这个原因，Elasticsearch为字段数据缓存提供了多种数据格式，分别针对不同的数据类型。

用户可以通过设置 `format` 属性来设置存储在字段数据缓存中数据的格式。可参考下面这个例子：

```
"tag" : {
  "type" : "string",
  "fielddata" : {
    "format" : "paged_bytes"
  }
}
```

现在，让我们来看看有哪些可用的数据格式。

（1）基于字符串的字段

针对基于字符串的字段，Elasticsearch 为字段数据缓存提供了 3 种数据格式。它们分别是：`paged_bytes`、`fst`、`doc_values`。默认格式是 `paged_bytes`，此时顺序存储词项所出现的位置，将文档映射到词项，此时数据存储在内存中。第 2 种格式是 `fst`，此时字段数据缓存中的数据存储在一种叫作 Finite State Transducer (FST 细节可参考 http://en.wikipedia.org/wiki/Finite_state_transducer) 的数据结构中。这种数据格式比默认的 `paged_bytes` 更节省内存，但是性能更慢。第 3 种数据格式为 `doc_values`，使用该格式时会在索引期计算字段数据缓存，数据存放在索引文件中。这种数据格式内存使用少，速度与默认格式相近。但是缺点是不能存储分词字段的数据，另外也不支持字段数据过滤。

（2）数值类型字段

针对数值类型的字段，为字段数据缓存提供了两种格式。默认格式为 `array`，此时数据在内存中以数组形式存储。第 2 种格式为 `doc_values`，此时使用 `doc values` 存储字段数据。这意味着，使用该格式时会在索引期计算字段数据缓存，数据存放在索引文件中，同时也不支持字段数据过滤。

（3）基于地理位置信息的字段

基于 `geo` 坐标的字段，与数值类型字段类似，默认格式为 `array`，坐标的经纬度都存储为数组类型。该字段也支持 `doc_values` 格式，使用 `doc values` 存储字段数据。当然，使用 `doc_values` 格式时，不支持字段数据过滤。

6. 字段数据加载

除了前面介绍的，Elasticsearch 还允许用户配置字段数据缓存的加载方式。前面提到过，默认情况下，字段数据缓存会在第一次使用数据时加载数据，即查询第一次使用到非倒排数据的时候。可以改变这种行为，方法是在查询中包含 `loading` 属性，将其值设置为 `eager`。这样设置会驱使 Elasticsearch 更主动地加载数据，一旦有数据更新，就会自动加载到缓存中。例如，如果您想为字段数据缓存配置主动加载 `tag` 字段数据，可按如下方式配置：

```
"tag" : {
  "type" : "string",
  "fielddata" : {
    "loading" : "eager"
  }
}
```

将 format 属性值设置为 disable，可禁用字段数据缓存的数据加载。例如，想禁止字段数据缓存加载 tag 字段数据，可按下面方式修改配置：

```
"tag" : {
  "type" : "string",
  "fielddata" : {
    "format" : "disabled"
  }
}
```

请记住，这样定义的字段，对于那些需要非倒排数据的功能是无效的。

6.6.3 查询分片缓存

Elasticsearch 1.4.0 中引入了一种新的缓存（查询分片缓存），可以帮助提高查询性能。其原理是为每个分片缓存本地查询结果。如果读者还记得的话，当 Elasticsearch 执行一个查询，查询会被发送给所有相关的分片，然后在每个分片上执行查询。然后查询结果会被发送至接收查询的节点进行合并。而分片缓存的是分片级的局部查询结果。



注意 在本书撰写的时候，仅缓存 search_type 指定的那些查询的相关计数。因此，查询返回的文档并不会被缓存，而每个分片返回的查询命中文档的个数、聚合、查询建议将会被缓存，这样也能在一定程度上帮助提升查询速度。这种状况将在 Elasticsearch 的未来版本中改变。

读者需注意，查询分片缓存默认是被禁用的。不过有两个方法开启它。第 1 种方法是通过在索引设置中添加 index.cache.query.enable 属性，并将其值设置为 true。或者用下面的实时命令更新索引设置：

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
  "index.cache.query.enable" : true
}'
```

第 2 种方法是每个请求中开启查询分片缓存。可通过在每个查询中设置名为 query_cache 的 URI 参数来实现。读者需注意，此时传递进来的参数将会覆盖索引级的同名设置。下面是一个查询请求的范例：

```
curl -XGET
'localhost:9200/books/_search?search_type=count&query_cache=true' -d
'{
  "query" : {
    "match_all" : {}
  },
  "aggregations" : {
    "tags" : {
      "terms" : {
        "field" : "tag"
      }
    }
  }
}'
```

值得一提的是，查询分片缓存有一大优点，它会自动失效及更新。当分片内容变更时，Elasticsearch 会自动更新缓存内容，因此缓存或未缓存查询的结果都是一样的。

设置查询分片缓存

默认情况下，Elasticsearch 集群中每个节点最多将 1% 堆内存用于查询分片缓存。这意味着一个节点上的所有索引的查询分片缓存最多能使用该节点的 1% 的堆内存。可通过设置配置文件 Elasticsearch.yml 中的 `indices.cache.query.size` 属性来调整这个百分比。

除此之外，也可以设置缓存的超时时间，这个可以通过设置 `indices.cache.query.expire` 属性来实现，例如，也许您想将超时时长设置为 60 分钟，那么将该属性值设置为 60m 即可。

6.6.4 使用 circuit breaker

因为查询会给 Elasticsearch 资源带来很大的压力，因此提供了一个叫作 circuit breaker 的功能用于限制某些特定功能使用过多的内存。Elasticsearch 会估算内存使用量，必要的时候（内存使用量到达某个阈值）会拒绝执行查询。现在我们来看看有哪些可用的 circuit breaker。

1. 字段数据 circuit breaker

如果某个查询的内存使用估算值高于预定值，字段数据 circuit breaker 将拒绝该查询执行。默认情况下，Elasticsearch 将 `indices.breaker fielddata.limit` 属性值设置为 60%，这意味着最多 JVM 堆内存的 60% 能用于字段数据缓存。

可以设置一个倍增系数，Elasticsearch 可以结合 `indices.breaker fielddata.overhead` 属性来估算内存使用量（内存估计量将会乘以该系数作为阈值）。默认情况下，该系数为 1.03。

**注意**

请记住，在 Elasticsearch 1.4.0 版本之前，`indices.breaker fielddata.limit`，被称为 `indices.fielddata.breaker.limit`。而 `indices.breaker.fielddata.overhead` 属性被称为 `indices.fielddatabreaker.overhead`。

2. request circuit breaker

Elasticsearch 1.4.0 中引入了 request circuit breaker，允许用户配置当总体内存使用的估计量高于 `indices.breaker.request.limit` 属性值时拒绝执行查询（阈值设置为 JVM 默认配置的堆内存的 40%）。

与字段数据 circuit breaker 类似，可设置 `indices.breaker.request.overhead` 属性值，默认为 1。

3. total circuit breaker

除了前面提到的这些 circuit breaker，Elasticsearch 1.4.0 中还引入了 total circuit breaker，该概念指的是所有 circuit breaker 可用内存之和。可以通过设置 `indices.breaker.total.overhead` 属性值来设置它，默认为 JVM 堆内存值的 70%。

**注意**

请注意，对于一个正常工作的集群，所有的 circuit breaker 可以通过集群配置更新 API 动态修改。

6.6.5 清除缓存

我们在前面提到过，有时候清除缓存很关键。Elasticsearch 允许我们使用 `_cache` REST 端点来清除缓存，我们现在就来讨论它的用法。

1. 单一索引缓存、多索引缓存和全部缓存的清除

清空全部缓存的最简单的做法是执行下面的命令：

```
curl -XPOST 'localhost:9200/_cache/clear'
```

当然，就像我们已经习惯了的那样，我们可以选择清空一个或多个索引的缓存。例如，如果我们想清除 `mastering` 索引的缓存，可以执行下面的命令：

```
curl -XPOST 'localhost:9200/mastering/_cache/clear'
```

如果我们想同时清除 `mastering` 和 `books` 索引的缓存，应该执行下面的命令：

```
curl -XPOST 'localhost:9200/mastering,books/_cache/clear'
```

2. 清除特定缓存

默认情况下，当有缓存清除请求到达，Elasticsearch 会清除所有的缓存。除了前面提到

的清除缓存的方法，我们也可以只清除一种指定类型的缓存。下面列出的是可以被单独清除的缓存类型。

- ❑ filter：这类缓存可以通过设置 filter 参数为 true 来清除。为了避免这种缓存被清除，我们需要设置 filter 参数为 false。请记住，此类缓存清理并不会立即生效，它会被 Elasticsearch 调度到下一个 60 秒内执行。
- ❑ field data：这类缓存可以通过设置 field_data 参数为 true 来清除。为了避免这种缓存被清除，我们需要设置 field_data 参数为 false。
- ❑ parent-child 关系：为了清除缓存中代表 parent-child 关系的 ID，可设置 id_cache 参数值为 true。设置该参数值为 false 将会防止缓存被清除。
- ❑ shard query：为了清除 shard query 缓存，可设置 query_cache 参数值为 true。设置该参数值为 false 将会防止缓存被清除。

例如，如果我们希望清除 mastering 索引的字段数据缓存，但是留下 filter 缓存和 shard query 缓存，则可以执行下面的命令：

```
curl -XPOST
'localhost:9200/mastering/_cache/clear?field_data=true&filter=false&
query_cache=false'
```

6.7 小结

在本章，我们学习了如何使用不同的相似度方法来改变 Lucene 的评分方式。同时也了解了通过 codec 改写倒排索引的写入格式。除此之外，还介绍了准实时搜索和实时读取及刷新对 Elasticsearch 的意义。另外，还讨论了如何按需配置事务日志及 I/O 子系统节流。最后介绍了索引的段合并、合并策略及调度，以及合并过程的可视化。最后，我们探讨了 Elasticsearch 的缓存功能。

在下一章，我们将近距离观察 Elasticsearch 的系统管理功能，也将了解节点发现与集群恢复功能，此外还会介绍用户友好的 Cat API。另外，还将介绍如何备份我们的索引，什么是联合搜索，以及如何在多个集群中索引和搜索数据。

管理 Elasticsearch

在上一章中，我们讨论了如何通过使用不同的相关性方法来调整 Lucene 的打分。我们以近乎准实时的方式索引和搜索数据，学习了如何 {flush} 和 {refresh} 数据。我们配置了事务日志和节流 IO 子系统，谈到了分段合并以及如何将它可视化。

这一章会讲到 Elasticsearch 配置和 ES 1.0 版及后续版本引入的新功能。到本章结束时，将涵盖以下内容：

- ☐ 配置发现和恢复模块
- ☐ 使用 Cat API 以人类可读的方式洞察集群状态
- ☐ 备份 / 还原功能
- ☐ 联盟搜索

7.1 发现和恢复模块

当你启动 ES 节点时，ES 最先做的事情之一就是查找一个拥有相同集群名称且在网络上可见的主节点。如果找到了，这个新启动的节点就加入那个已经存在的集群。如果没找到，这个新节点就选自己成为主节点（当然了，如果你的配置允许它这么做的话）。发现节点和组成集群的过程叫作发现（discovery）。负责发现的模块有两个作用：选主节点和发现集群的新节点。

集群建立后，一个称为恢复（recovery）的过程就开始了。在恢复过程中，ES 从网关读取元数据和索引，并且准备好保存在那里的需要使用的分片。主分片恢复完成之后，ES 就

应该可以响应外部请求了。同时，如果存在副本的话，ES 将继续恢复其他的副本。

在这一节里，我们会深入了解这两个模块，并讨论 ES 提供了哪些配置以及修改这些配置后会有哪些影响。



注意 在已经发布的《Elasticsearch Server, Second Edition》中我们也提到了发现和恢复模块，本节内容是对其的扩展。

7.1.1 发现模块的配置

就如我们多次提到的，ES 被设计成在分布式的环境中工作。这是 ES 与其他开源搜索和分析解决方案相比最主要的区别。基于这个设定，非常容易在分布式环境中组建 ES 集群，并不需要强制安装额外的软件。默认情况下，ES 假定集群由声明了相同集群名称且可以使用组播来相互通信的节点自动组建。这允许我们在同一个网络中组建多个相互独立的集群。

发现模块有多个实现，下面我们来看看具体是哪些。

Zen 发现

Zen 发现是 ES 里承担发现职责的默认实现，默认有效。Zen 发现的默认配置使用组播来发现其他节点。这是一个很方便的解决方案，只要启动一个 ES 节点就可以了（新节点会自动加入拥有相同集群名称的集群，并对集群中其他节点可见）。这一模式非常适合开发阶段，因为你无需关心如何配置。不过，我们不建议在生产环境中使用它。依赖于集群名称是很方便的，但也会导致潜在的问题和错误，例如节点的意外加入。有时组播会由于各种原因不可用，或者你由于这些提及的原因而不想使用组播。对于一个大规模的集群，组播可能会产生大量不必要的通信，这是另一个我们不应该在生产环境下使用 Zen 发现的理由。

在这些情形下，Zen 发现允许我们使用单播模式。当使用单播 Zen 发现时，集群外的节点会发送一个 Ping 请求到所有配置中指定的地址。通过这种方式，它通知所有指定节点其已准备好成为集群中的一员，要么加入一个现有集群，要么组建一个新的集群。当然了，节点加入集群后会获得集群的拓扑信息，但是初始的请求仅发送给了指定的主机。有一点需要注意一下，即使在使用单播 Zen 发现时，节点也需要有与其他节点相同的集群名。



注意 如果你想了解关于 ping 方法的单播和组播的更多不同之处，可以查看这些地址：<http://en.wikipedia.org/wiki/Multicast> 和 <http://en.wikipedia.org/wiki/Unicast>。

如果你想了解组播 Zen 发现的更多配置，我们就来看看它们。

(1) 组播 Zen 发现配置

Zen 发现模块的组播部分对外提供如下配置。

- ❑ `discovery.zen.ping.multicast.address` : 用来通信的网络接口, 可以是地址或者用接口名称来指定。默认是所有可用的网络接口。
- ❑ `discovery.zen.ping.multicast.port`: 用来通信的端口号, 默认值是 54328。
- ❑ `discovery.zen.ping.multicast.group`: 组播消息需要发送到的地址, 默认是 224.2.2.4。
- ❑ `discovery.zen.ping.multicast.buffer_size`: 组播消息使用的缓冲区大小, 默认是 2028。
- ❑ `discovery.zen.ping.multicast.ttl`: 组播消息的生存时间默认值是 3。消息包每次通过一个路由器, TTL 就减一。这可以限制消息能传输到的区域。路由器有可以配置的阈值来对比 TTL, 这使得 TTL 的值可能不等于消息包可以通过的路由器的个数。
- ❑ `discovery.zen.ping.multicast.enabled` : 是否开启组播, 默认值是 `true`。设置为 `false` 时会关闭组播。如果你想使用单播 Zen 发现的话, 你应该关闭组播。

(2) 单播 Zen 发现配置

Zen 发现模块的单播部对外提供如下配置。

- ❑ `discovery.zen.ping.unicast.hosts`: 集群初始节点列表。可以是一个节点列表或者数组。每个节点可以配置一个名称或者 IP 地址, 还可以加上一个端口号或者端口范围。例如: `["master1", "master2:8181", "master3[80000-81000]"]`。通常节点列表不必包含集群中全部的节点, 因为一旦节点连接上了列表中的任意节点, 就会被告知集群中所有节点的信息。
- ❑ `discovery.zen.ping.unicast.concurrent_connects`: 单播发现使用的最大并发链接数, 默认 10 个。如果初始连接阶段有大量的节点需要连接, 你应该调高这个默认值。

7.1.2 主节点

主节点会监控和管理集群中的其他节点, 除了连接其他节点外, 选择主节点也是发现模块的主要用途之一。选择的过程被称为主节点选举 (master election)。无论存在多少主节点, 每个集群在给定的时间内都只会会有一个活动的主节点。如果集群中有多个主节点, 那么当原来的活动主节点宕机并从集群中删除时, 可以从中选举新的活动主节点。

1. 配置主节点和数据节点

ES 默认允许所有节点成为主节点和数据节点。然而, 在特定的情况下, 你可能会想要只承载数据或者处理查询的工作节点, 以及仅作为集群管理者的主节点。其中的一种情况是处理大量的数据, 这时数据节点需要尽可能高的性能, 不应该被主节点的职能所延误。

(1) 配置只持有数据的节点

为了设置节点只持有数据，我们需要告诉 ES 我们不希望这个节点成为主节点。为了达到这个目的，我们需要在 `Elasticsearch.yml` 文件中添加以下的属性：

```
node.master: false
node.data: true
```

(2) 配置只作为主节点的节点

为了设置节点不持有数据并且仅作为主节点，我们需要告诉 ES 我们不想这个节点持有数据。为了达到这个目的，我们需要在 `Elasticsearch.yml` 文件中添加以下的属性：

```
node.master: true
node.data: false
```

(3) 配置只处理查询请求的节点

对于一个足够大的集群，配置一些仅聚合其他节点查询结果的节点是明智的。这些节点应该配置为非数据节点、非主节点。于是它们的 `Elasticsearch.yml` 文件中应该有以下的属性：

```
node.master: false
node.data: false
```



注意 `node.master` 和 `node.data` 属性默认值是 `true`，但为了使配置清晰，我们倾向于显式地设置它们。

2. 主节点选举的相关配置

我们已经在《*Elasticsearch Server, Second Edition*》中写过关于主节点选举的配置，但是这个主题非常重要，所以我们决定再刷新一下其相关的知识。

假设你拥有一个由 10 个节点组成的集群。它一直工作得很好，直到有一天，网络出现故障，有 3 个节点从集群中断开连接了，但这 3 个节点仍然可以相互访问。由于 Zen 发现和主节点选举进程的存在，这些脱离集群的节点会选举出一个新的主节点，于是就产生了两个同名的集群和两个主节点。这种情形被称为脑分裂（split-brain），你必须尽可能地避免出现这种情形。当发生脑分裂时，将会存在两个或者更多的集群，直至网络或者其他问题被修复。如果你在这个期间索引数据，那么，当集群从脑分裂中恢复时，会出现数据丢失和不可恢复的情况。

为了避免脑分裂的出现，或者至少降低其出现的概率，Elasticsearch 提供了 `discovery.zen.minimum_master_nodes` 属性。这个属性定义了为组建集群至少需要的相互连接的候选主节点数量。现在回到我们前面讨论的集群，当我们设置 `discovery.zen.minimum_master_nodes` 属性的值为集群中一半的节点数加 1 时，对于我们的集群来说就是 6，那么我们就只会 1 个集群。为什么会这样呢？因为在网络出现故障前，我们有 10 个节点，多于 6 个，这些节

点会组建一个集群。当有 3 个节点断开时，这个集群还会正常运行。然而，由于有 3 个节点断开，而且 3 小于 6，这 3 个节点不被允许选举一个新的主节点，它们会等待重新连接上初始的那个集群。

Zen 发现故障检测和配置

Elasticsearch 在工作时会运行两个检测进程。第 1 个进程是由主节点发送 ping 请求到集群中的其他全部节点，检测它们是否可用。第 2 个进程是相反的过程，每个节点都发送 ping 请求到主节点，检测主节点是否在运行并履行其职责。然而，如果我们有一个缓慢的网络或者我们的节点处在不同的地点，默认的配置就可能不够充分了。于是 Elasticsearch 的发现模块提供了 3 个我们可以修改的属性。

- `discovery.zen.fd.ping_interval`: 定义节点多久向目标节点发送一次 ping 请求，默认 1 秒。
- `discovery.zen.fd.ping_timeout`: 定义节点在接到 ping 响应前会等待多久，默认 30 秒。

如果你的节点被 100% 的使用或者网络较慢，你可以考虑增加等待时间。

- `discovery.zen.fd.ping_retries`: 定义在目标节点被认为不可用前最大的 ping 请求重试次数，默认 3 次。如果你的网络丢包严重，你可以调高重试次数，或者你可以修复你的网络。

还有一点我们想说一下。主节点是唯一可以改变集群状态的节点。为了实现一个恰当的集群状态更新序列，Elasticsearch 的主节点每次处理一个集群状态更新请求，在本地更新，然后发送请求给其他节点，以使这些节点能够同步状态。主节点会在指定的时间内等待其他节点的响应，如果超时或者全部的节点都返回了当前的确认信息，它会继续执行下一个更新集群状态的请求。为了修改主节点等待回应的时间，你可以修改 `discovery.zen.publish_timeout` 属性，默认是 30 秒。在一个繁忙的网络中工作的大型集群可能会需要调高这个属性。

3. 亚马逊 EC2 发现

亚马逊除了销售商品之外，还销售一些流行的服务，例如按使用量付费的存储空间和计算能力。被称为亚马逊弹性计算云（EC2）的服务提供服务器实例，当然它们可以被用来安装和运行 Elasticsearch 集群（也能用来做其他事情，因为它们就是普通的 Linux 服务器）。为了应对流量的变化或者提高计算能力而按照你需要的实例数量来付费是很便利的，当流量小的时候你可以关闭不必要的实例。Elasticsearch 可以很好地运行在 EC2 上，但由于环境的特性，一些功能的工作方式会有所不同。其中一个功能就是发现，因为亚马逊 EC2 不支持组播发现。当然我们可以切换到单播发现，但有时我们想要自动发现节点，而使用单播我们至少需要配置初始主机列表。然而，我们有一个替代，我们可以使用亚马逊 EC2 插件，它使用 EC2 的 API 来实现单播和组播发现。



注意 确保在配置 EC2 实例期间，实例间是可以通信的（默认通过 9200 和 9300 端口），这对 Elasticsearch 节点可以相互通信是重要的，组建集群需要它们能够相互通信。当然这个通信依赖于 `network.bind_host` 和 `network.publish_host`（或者 `network.host`）的配置。

（1）EC2 插件的安装

安装 EC2 插件同安装大多数插件一样的简单，为了安装这个插件，我们需要执行以下的命令：

```
bin/plugin install elasticsearch/elasticsearch-cloud-aws/2.4.0
```

（2）EC2 插件的通用配置

为了让 EC2 发现能够工作，EC2 插件提供了一些需要我们配置的属性。

❑ `cluster.aws.access_key`：亚马逊 `access_key`，身份凭据之一，你可以在亚马逊配置面板中找到它们。

❑ `cluster.aws.secrete_key`：亚马逊 `secrete_key`，同前面提到的 `access_key` 类似，你可以在 EC2 的配置面板中找到它。

最后要做的就是通知 Elasticsearch 我们想要使用一个新的发现类型，并且关闭组播。可以通过设置属性 `discovery.type` 属性为 `ec2` 来通知 Elasticsearch。

（3）EC2 插件的可选配置

前面提到的配置已经足够运行 EC2 发现了，但是为了控制 EC2 发现的行为，Elasticsearch 提供了附加的配置。

❑ `cloud.aws.region`：地区在连接亚马逊 Web 服务时用到。你可以选择你的实例所驻留的地区作为这项的值，例如 `eu-west-1` 代表冰岛。在本书写作时可选的值是 `eu-west`、`sa-east`、`us-east`、`us-west-1`、`us-west-2`、`ap-southeast-1` 和 `ap-southeast-1`。

❑ `cloud.aws.ec2.endpoint`：如果你使用 EC2 的 API 服务，除了定义地区，你还可以提供一个 AWS 端点的地址，例如，`ec2.eu-west-1.amazonaws.com`。

❑ `cloud.aws.protocol`：这是插件用来连接亚马逊 web 服务的协议。Elasticsearch 默认使用 HTTPS 协议（这意味着配置属性值为 `https`）。我们也可以通过设置属性值为 `http` 来改变这个行为，这样插件就会使用不加密的 HTTP 协议了。我们也可以通过配置 `cloud.aws.ec2.protocol` 和 `cloud.aws.s3.protocol` 属性来覆盖每个服务的 `cloud.aws.protocol` 配置，可选的值也是 `https` 和 `http`。

❑ `cloud.aws.proxy_host`：Elasticsearch 允许我们使用一个代理来连接 AWS 端点。`cloud.aws.proxy_host` 属性应当被设置为使用的代理的地址。

❑ `cloud.aws.proxy_port`：AWS 端点代理监听的端口号。

❑ `cloud.aws.ec2.ping_timeout`：等待发送到其他节点的 ping 请求的被响应的的时间，默

认 3s。超过这个时间，没有响应的节点会被认为已经宕机并从集群中移除。在遇到网络问题或者我们有大量 EC2 节点时，调高这项配置是有意义的。

(4) EC2 节点扫描配置

我们想要提及的最后一组配置在组建工作在 EC2 环境下的集群时非常重要。它们能够过滤运行在亚马逊云计算网络上的可用 Elasticsearch 节点。EC2 插件提供以下属性来帮助控制它的行为。

- ❑ `discovery.ec2.host_type`：这个配置允许我们选择用来与集群中其他节点通信时使用的主机类型。我们可以使用的值是 `private_ip`（默认值，使用私有 IP 进行通信）、`public_ip`（使用公开 IP 进行通信）、`private_dns`（使用私有主机名进行通信）和 `public_dns`（使用公开的主机名进行通信）。
- ❑ `discovery.ec2.groups`：这项是一个用逗号分隔的安全组列表。只有组内的节点才能够被发现并包含进集群中。
- ❑ `discovery.ec2.availability_zones`：数组或者逗号分隔的可用地区列表。只有指定地区的节点才能够被发现并包含进集群中。
- ❑ `discovery.ec2.any_group`：默认是 `true`，设置为 `false` 时，会强制 EC2 插件仅发现那些匹配全部安全组的节点。默认值仅要求匹配一个安全组。
- ❑ `discovery.ec2.tag`：这是一组 EC2 相关配置的前缀。当你启动亚马逊 EC2 实例时，你可以定义标签来描述实例的用途，例如自定义的名称或者环境类型。然后，你用这些定义的标签来限制节点发现。假设你定义了一个名称为 `environment`，值是 `qa` 的标签。在配置文件中你可以做以下的配置。
 - `discovery.ec2.tag.environment:qa`（和只有定义了这个标签的实例在做节点发现时才会被考虑进去）。
 - `cloud.node.auto_arrrtributes`：当设置为 `true` 时，（Elasticsearch 会添加 EC2 相关的节点属性到节点属性中，例如地区和组。并允许我们在 Elasticsearch 的分片部署和分片替换时使用。你可以在第 5 章的分布式索引架构中找到更多的关于分片替换的信息。

4. 其他节点发现方式

Zen 发现和 EC2 发现并不是仅有的发现类型。还有两种发现类型被 Elasticsearch 团队所开发并维护，它们如下所示。

- ❑ Azure 发现：<https://github.com/Elasticsearch/Elasticsearchcloud-azure>
- ❑ Google Compute Engine discovery：<https://github.com/Elasticsearch/Elasticsearchcloud-gce>

除了这些，还有一些社区提供的发现实现，例如支持早期 Elasticsearch 版本的

Zookeeper 发现 (<https://github.com/sonian/Elasticsearch-zookeeper>)。

7.1.3 网关和恢复模块的配置

网关模块允许我们存储 Elasticsearch 正常运行所需的全部数据。这意味着不仅存储 Apache Lucene 的索引数据, 还存储所有的元数据 (例如关于索引分配的相关配置), 以及每个索引的映射信息。每当集群的状态改变时, 例如, 当分配属性被修改了, 集群的状态都会通过网关模块持久化。当集群启动时, 集群的状态会从网关模块加载并应用在集群上。

当为不同的节点配置了不同的网关类型时, 索引会使用其所在节点上的网关配置。如果索引的状态不应该通过网关模块来保存, 你需要显式地设置索引网关类型为 `none`。

1. 通过网关来恢复的过程

让我们说得更清楚些, 恢复过程加载通过网关模块存储的数据以使 Elasticsearch 正常工作。每当集群整体重启发生时, 恢复过程就会启动, 加载所有我们提到的相关信息: 元数据、映射和索引。当恢复过程启动时, 主分片 (primary shard) 会首先初始化, 然后根据副本 (replica) 的状态, 副本会使用网关数据, 或者当它们同主分片不同步时使用拷贝自主分片的数据。

Elasticsearch 允许我们配置何时需要使用网关模块恢复集群数据。我们可以告诉 Elasticsearch 在开始恢复过程前等待一定数量的候选主节点或者数据节点加入集群。然而, 需要注意的是, 在集群完成恢复前, 其上的所有操作都是不被允许的。这样做的目的是防止修改冲突。

2. 相关配置属性

在开始讨论配置之前, 还想说一件事情。你知道 Elasticsearch 可以有不同的角色, 它们可以是数据节点 (只持有数据), 可以是主节点, 或者仅作为请求处理节点 (既不持有数据, 也不是主节点)。记住这些之后我们来看看可以修改的网关配置。

- ❑ `gateway.recovery_after_nodes`: 集群中存在多少个节点后才启动恢复过程。例如, 设为 5, 那么至少需要 5 个节点加入才会开始恢复过程, 无论它们是数据节点还是主节点。
- ❑ `gateway.recovery_after_data_nodes`: 集群中存在多少个数据节点后才启动恢复过程。
- ❑ `gateway.recovery_after_master_nodes`: 集群中存在多少个主节点后才启动恢复过程。
- ❑ `gateway.recovery_after_time`: 当前面的条件满足后, 等待多少时间才开始恢复过程。例如设置为 5m, 当定义好的条件满足后, 再过 5 分钟才会开始恢复过程。从 Elasticsearch 1.3.0 开始, 默认是 5 分钟。

假设我们的集群有 6 个节点, 其中 4 个是数据节点。我们还有一个由 3 个分片构成的索引分布在集群中。最后两个节点是主节点, 不持有数据。我们希望配置恢复过程在 4 个

数据节点加入后延迟 3 分钟开始。那么我们的配置就会是下面的样子：

```
gateway.recover_after_data_nodes: 4
gateway.recover_after_time: 3m
```

3. 对于节点的期望

除了我们已经提到的属性外，还有一些属性可以强制开始 Elasticsearch 的恢复过程。它们如下所示：

- ❑ `gateway.expected_nodes`：立即开始恢复过程前集群中必须存在的节点数。如果你不希望恢复过程延迟启动，建议设置这个属性为足够组成集群的节点数（或者至少是大多数），因为这样能够确保集群恢复到最近的状态。
- ❑ `gateway.expected_data_nodes`：立即开始恢复过程前集群中必须存在的数据节点数。
- ❑ `gateway.expected_master_nodes`：立即开始恢复过程前集群中必须存在的主节点数。

现在回到前面的例子。我们希望在全部的 6 个节点在线时启动恢复过程。所以，除了前面的配置外，我们可以加上以下的配置：

```
gateway.expected_nodes: 6
```

完整的配置如下：

```
gateway.recover_after_data_nodes: 4
gateway.recover_after_time: 3m
gateway.expected_nodes: 6
```

上面的配置意味着，一旦有 4 个数据节点在线，恢复过程会延迟 3 分钟后启动；而当 6 个节点在线时会立刻启动，不论它们是数据节点还是主节点。

4. 本地网关

随着 0.20 版本 Elasticsearch 的发布（以及一些 0.19 版之后的版本），除了本地网关外的其他类型的网关都被弃用了。建议你不要再使用它们了，因为它们会在 Elasticsearch 将来的版本中被移除。现在它们还在，但是如果你想避免重新索引全部的数据，你应该只使用本地网关。这也是为什么我们不讨论其他类型的网关。

本地网关使用节点上可用的本地存储来保存元数据、映射和索引。为了使用本地网关和节点上可用的本地存储，需要有足够的磁盘空间来容纳数据。

本地网关持久化数据的方式不同于其他的类型的网关（已经弃用）。本地网关在写入数据时是以同步的方式进行的，这样是为了确保在写入过程中不会丢失数据。



注意 想要设置使用的网关类型，可以设置 `gateway.type` 属性，默认是 `local`。

关于 Elasticsearch 的本地网关还有一点我们没有谈到，就是悬空索引（dangling indices）。当一个节点加入集群，节点上存在的但不在当前集群中的分片和索引也会被包含

进集群中。这类索引就被叫作悬空索引。我们可以选择 Elasticsearch 对待它们的方式。

Elasticsearch 提供了 `gateway.local.auto_import_dangling` 属性，可以接受的值有 `yes`（默认值，表示导入所有的悬空索引到集群中）、`close`（导入悬空索引到集群中，但是置为关闭状态）、`no`（删除悬空索引）。当配置为 `no` 时，我们还可以配置 `gateway.local.dangling_timeout` 属性（默认 2 小时）来指定在删除悬空索引时 Elasticsearch 会等待多久。悬空索引功能在我们重启 Elasticsearch 旧节点，且不希望老索引被包含进集群中时能有所帮助。

5. 恢复过程的底层配置

我们讨论过可以通过网关来控制 Elasticsearch 恢复过程的行为，但是，除此之外，Elasticsearch 也允许我们直接配置恢复过程如何工作。在谈到分片分配（5.3 节）时，我们已经提到过一些恢复配置选项，但是，我们认为在专注网关和恢复的章节里讨论一下这些配置是有必要的。

（1）集群级别的恢复配置

恢复过程的配置多数都在集群级别给定，允许我们设置恢复模块工作时遵守的通用规则。这些配置如下。

- ❑ `indices.recovery.concurrent_streams`：在从数据源恢复一个分片时可以同时打开的流的数量，默认为 3 个。这个值越高，带给网络层的压力就越大，不过，依赖于网络使用和吞吐量，恢复过程可能更快些。
- ❑ `indices.recovery.max_types_per_sec`：在恢复分片时每秒可以传输的最大数据量，默认为 20MB。如果要需要传输限制，可以设置为 0。同并发流的数量类似，这个属性可以用来控制恢复过程对于网络的使用。设置为更高的值可带来更高的网络使用和更短的恢复时间。
- ❑ `indices.recovery.compress`：恢复过程在传输数据时是否压缩数据，默认为 `true`。设为 `false` 可以降低 CPU 的压力，但是会造成网络传输数据量的加大。
- ❑ `indices.recovery.file_chunk_size`：从源分片拷贝数据时数据块的大小，默认是 512KB，当开启了压缩选项时数据块会被压缩。
- ❑ `indices.recovery.translog_ops`：恢复过程的一次请求里在分片间传输的事务日志的行数，默认为 1000。
- ❑ `indices.recovery.translog_size`：从源分片拷贝事务日志时使用的数据块的大小，默认为 512KB，当开启了压缩选项时数据块会被压缩。



注意 在 0.90.0 版之前的 Elasticsearch 版本中，有一个 `indices.recovery.max_size_per_sec` 属性，现在已经弃用了，并建议使用 `indices.recovery.max_types_per_sec` 属性来替代。然而，如果你使用的是 0.90.0 之前的版本，可能需要用到这个。

前面提到的全部配置都能使用更新集群的 API 来设置，或者使用 Elasticsearch.yml 文件来配置。

(2) 索引级的恢复配置

除了前面提到的那些配置项，还有一个索引级的配置项。这个配置可以通过 Elasticsearch.yml 文件和更新索引的 API 来设置。这个配置项是 `index.recovery.initial_shards`。通常，只有在有特定数量的分片存在，并且可以被部署时，Elasticsearch 才会恢复一个分片。这个特定数量是指定索引的分片数量的一半加上 1。通过使用 `index.recovery.initial_shards` 配置，可以改变 Elasticsearch 将什么当作这个特定数量。可选的配置值如下。

- ❑ `quorum`: 50% 加 1 的分片存在且可部署。这项是默认值。
- ❑ `quorum-1`: 50% 的分片存在且可部署。
- ❑ `full`: 给定索引的全部分片存在且可部署。
- ❑ `full-1`: 给定索引的全部分片数减 1 个分片存在且可部署。
- ❑ 整数值: 任意整数，例如 1、2 和 5。指定需要存在的可部署分片数。例如，设置为 2 意味着至少需要存在 2 个可部署的分片，Elasticsearch 才会恢复这个索引的分片。

了解这个配置项是有意义的，但是在大多数情况下，默认值对于 Elasticsearch 的部署已经足够了。

7.1.4 索引恢复 API

介绍了索引恢复 API 后，我们将不再受限于仅仅观察集群的状态，类似以下内容：

```
curl 'localhost:9200/_cluster/health?pretty'
{
  "cluster_name" : "mastering_elasticsearch",
  "status" : "red",
  "timed_out" : false,
  "number_of_nodes" : 10,
  "number_of_data_nodes" : 10,
  "active_primary_shards" : 9,
  "active_shards" : 9,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 1
}
```

通过向 `_recovery` 端点发送 GET 请求（查询全部索引或者指定索引），我们可以得到索引恢复的状态。例如，我们看以下的请求：

```
curl -XGET 'localhost:9200/_recovery?pretty'
```

前面的请求会返回关于集群中所有分片的恢复相关的信息，包括正在进行和已经完成的。在我们的例子里如下：

```

{
  "test_index" : {
    "shards" : [ {
      "id" : 3,
      "type" : "GATEWAY",
      "stage" : "START",
      "primary" : true,
      "start_time_in_millis" : 1414362635212,
      "stop_time_in_millis" : 0,
      "total_time_in_millis" : 175,
      "source" : {
        "id" : "3M_ErmCNTR-huTqOTv5smw",
        "host" : "192.168.1.10",
        "transport_address" : "inet[/192.168.1.10:9300]",
        "ip" : "192.168.10",
        "name" : "node1"
      },
      "target" : {
        "id" : "3M_ErmCNTR-huTqOTv5smw",
        "host" : "192.168.1.10",
        "transport_address" : "inet[/192.168.1.10:9300]",
        "ip" : "192.168.1.10",
        "name" : "node1"
      },
      "index" : {
        "files" : {
          "total" : 400,
          "reused" : 400,
          "recovered" : 400,
          "percent" : "100.0%"
        },
        "bytes" : {
          "total" : 2455604486,
          "reused" : 2455604486,
          "recovered" : 2455604486,
          "percent" : "100.0%"
        },
        "total_time_in_millis" : 28
      },
      "translog" : {
        "recovered" : 0,
        "total_time_in_millis" : 0
      },
      "start" : {
        "check_index_time_in_millis" : 0,
        "total_time_in_millis" : 0
      }
    }, {
      "id" : 9,
      "type" : "GATEWAY",
      "stage" : "DONE",
      "primary" : true,
      "start_time_in_millis" : 1414085189696,

```

```
{
  "stop_time_in_millis" : 1414085189729,
  "total_time_in_millis" : 33,
  "source" : {
    "id" : "nNw_k7_XSOivvPCJLHVE5A",
    "host" : "192.168.1.11",
    "transport_address" : "inet[/192.168.1.11:9300]",
    "ip" : "192.168.1.11",
    "name" : "node3"
  },
  "target" : {
    "id" : "nNw_k7_XSOivvPCJLHVE5A",
    "host" : "192.168.1.11",
    "transport_address" : "inet[/192.168.1.11:9300]",
    "ip" : "192.168.1.11",
    "name" : "node3"
  },
  "index" : {
    "files" : {
      "total" : 0,
      "reused" : 0,
      "recovered" : 0,
      "percent" : "0.0%"
    },
    "bytes" : {
      "total" : 0,
      "reused" : 0,
      "recovered" : 0,
      "percent" : "0.0%"
    },
    "total_time_in_millis" : 0
  },
  "translog" : {
    "recovered" : 0,
    "total_time_in_millis" : 0
  },
  "start" : {
    "check_index_time_in_millis" : 0,
    "total_time_in_millis" : 33
  },
  .
  .
}
}
```

前面的响应中包含了 `test_index` 索引的两个分片的信息（为了方便观察，其他的分片信息被删除了）。我们可以看到一个分片正在恢复中（`"state": "started"`），另一个分片已经完成恢复了（`"state": "DONE"`）。我们可以看到大量的有关恢复过程的信息，这些信息是索引级别的信息，通过这些信息我们可以清楚地看到 Elasticsearch 处在怎样的状态。我们也可以

通过在请求中加上 `active_only=true` 参数来限制只返回处在恢复中的分片。例如：

```
curl -XGET 'localhost:9200/_recovery?active_only=true&pretty'
```

如果我们想获得更详细的信息，可以在请求中加上 `detailed=true` 参数，就像下面这样：

```
curl -XGET 'localhost:9200/_recovery?detailed=true&pretty'
```

7.2 使用人类友好的Cat API

Elasticsearch 管理 API 非常广泛，涵盖了 Elasticsearch 架构的几乎每个部分，从有关 Lucene 的低层信息，到关于集群节点和其健康状态的高层信息。所有的这些信息都能通过 Elasticsearch 提供的 Java API 或者 REST API 来得到。然而，这些信息是 JSON 格式的，并且，返回的数据有时不进行进一步的解析是难以分析的。例如，试着在你的 Elasticsearch 集群上执行以下请求：

```
curl -XGET 'localhost:9200/_stats?pretty'
```

对于本地只有一个节点的集群，Elasticsearch 返回了如下的信息（做了大量的删减，完整的响应可以在随书提供的 `stats.json` 文件中找到）：

```
{
  "_shards" : {
    "total" : 60,
    "successful" : 30,
    "failed" : 0
  },
  "_all" : {
    "primaries" : {
      .
      .
      .
    },
    "total" : {
      .
      .
      .
    }
  },
  "indices" : {
    .
    .
    .
  }
}
```

如果你查看 `stats.json` 文件，你会看到响应有大约 1 350 行。对于人类来说，不进行进一步的解析这是很难分析的。由于这个原因，Elasticsearch 给我们提供了更加人性化友好的

Cat API。Cat API 以简单的文本、表格的形式来返回数据，并且还提供常用的聚合信息，避免了对数据的进一步处理。



注意 记得曾告诉过你，Elasticsearch 允许你获取的信息并不限于 JSON 格式吗？如果你忘记了，请试着在请求中加上 `format=yaml` 参数。

7.2.1 基础知识

Cat API 的基础端点非常明显，就是 `/_cat`，没有任何参数，它会显示所有可用的端点。我们可以通过运行以下的命令来测试一下：

```
curl -XGET 'localhost:9200/_cat'
```

Elasticsearch 的响应应该跟下面的类似或者相同（依赖于你的 Elasticsearch 的版本）：

```
=^.^=  
/_cat/allocation  
/_cat/shards  
/_cat/shards/{index}  
/_cat/master  
/_cat/nodes  
/_cat/indices  
/_cat/indices/{index}  
/_cat/segments  
/_cat/segments/{index}  
/_cat/count  
/_cat/count/{index}  
/_cat/recovery  
/_cat/recovery/{index}  
/_cat/health  
/_cat/pending_tasks  
/_cat/aliases  
/_cat/aliases/{alias}  
/_cat/thread_pool  
/_cat/plugins  
/_cat/fielddata  
/_cat/fielddata/{fields}
```

看一下 Elasticsearch 允许我们使用 Cat API 来获取哪些信息：

- ☐ 分片部署相关的信息
- ☐ 所有分片相关的信息（限定为特定的索引）
- ☐ 节点的信息，包括选出的主节点的信息
- ☐ 索引统计信息（限定为特定的索引）
- ☐ 段的统计信息（限定为特定的索引）
- ☐ 文档计数（限定为特定的索引）

- ❑ 恢复信息（限定为特定的索引）
- ❑ 集群健康情况
- ❑ 待执行任务
- ❑ 索引别名和指定别名所对应的索引
- ❑ 线程池配置
- ❑ 每个节点上安装的插件
- ❑ 字段数据缓存的大小和每个字段数据缓存的大小

7.2.2 使用 Cat API

让我们通过一个例子来开始使用 Cat API。我们以查看集群健康状态来开始。我们只需执行以下的命令：

```
curl -XGET 'localhost:9200/_cat/health'
```

Elasticsearch 对于上面命令的响应应该类似于以下内容：

```
1414347090 19:11:30 elasticsearch yellow 1 1 47 47 0 0 47
```

这非常清晰易读。因为它是以表格的形式，非常容易把这个响应用于 `grep`、`awk` 或者 `sed` 工具中。一旦你了解了响应内容的含义，可读性会更强。为了给每一列加上一个描述其意义的标题，我们只需要增加一个 `v` 参数，就像下面这样：

```
curl -XGET 'localhost:9200/_cat/health?v'
```

这次的响应跟我们前面看到的很像，但是现在有一个表头来描述每一列：

epoch	timestamp	cluster	status	node.total	node.data	shards
pri	relo	init	unassign			
1414347107	19:11:47	elasticsearch	yellow	1	1	47
47	0	0	47			

通用参数

每个 Cat API 都有自己的参数，但是有一些选项是它们共同拥有的。

- ❑ `v`：给响应添加一个表头，标明每列数据的名称。
- ❑ `h`：限制只显示选定的列（参见下节的内容）。
- ❑ `help`：显示这个特定端点可以显示的所有可能的列。显示这个特定端点的参数名、参数缩写和其描述信息。
- ❑ `bytes`：这是呈现字节量信息的格式。我们说过，Cat API 被设计为给人类使用，由此，这些值默认以人类可读的方式呈现，例如：3.5kB，或者 40GB。bytes 选项允许我们给所有的数字设置基数，因此排序或者对比数值就相对容易了。例如 `bytes=b` 表示所有的值是以 byte 为单位的，`bytes=k` 表示以 kB 为单位，以此类推。



注意

想查看每个 Cat API 端点的完整参数列表，可以参考 Elasticsearch 的官方文档，地址是 <http://www.Elasticsearch.org/guide/en/Elasticsearch/reference/current/cat.html>。

7.2.3 一些例子

当我们撰写本书时，Cat API 有 21 个端点。我们不想全部都介绍一下，那样只是对官方文档信息的重复，或者是对管理 API 章节的重复。但是在没有给出任何关于使用 Cat API 的例子前，我们还不想结束这个小节。因此，我们决定向你展示一下，与对比 Elasticsearch 标准的 JSON API 相比，使用 Cat API 获取信息有多么的容易。

1. 获取关于主节点的信息

第 1 个例子向你展示获得集群中哪个节点是主节点的信息有多么容易。通过调用 `/_cat/master` 端点，我们能够得到节点中哪个被选为了主节点。例如，我们执行以下的命令：

```
curl -XGET 'localhost:9200/_cat/master?v'
```

对于本地的 2 节点集群，Elasticsearch 的响应看起来是下面的样子：

id	host	ip	node
8gfdQlV-SxKB0uUxkjbxSg	Banshee.local	10.0.1.3	Siege

从响应中你可以看到，我们得到了哪个节点被选为主节点，我们能看到它的 ID、IP 地址和名称。

2. 获得关于节点的信息

`/_cat/nodes` 端点提供了集群中全部节点的信息。让我们看看在执行以下的命令后 Elasticsearch 会返回什么：

```
curl -XGET 'localhost:9200/_cat/nodes?v&h=name,node.role,load,uptime'
```

在上面的例子里，我们使用了从这个端点的大约 70 个选项中定制我们想要返回的信息的能力。我们现在只获得节点名称、角色（节点是数据节点还是客户端节点）、负载和运行时间。

Elasticsearch 的响应如下：

name	node.role	load	uptime
Alicia Masters	d	6.09	6.7m
Siege	d	6.09	1h

如你所见，`/_cat/nodes` 端点提供了请求的关于集群中节点的全部信息。

7.3 备份

对于管理员来说，其最重要的任务之一就是确保在系统出故障时没有数据会丢失。在

Elasticsearch 的设定中，它是一个健壮的、易于配置的由节点组成的集群，甚至能够在一些并发的故障中幸免。然而，即使是完美配置的集群在面对网络分割和网络分区时也是脆弱的，在一些罕见的情况下会造成数据损坏或者丢失。在这些情况下，能够从重建索引中拯救我们的，就只有从备份中恢复数据这唯一的办法了。

你可能已经知道我们要说什么了：Elasticsearch 提供的快照 / 还原功能。不过，就像我们之前说的，我们不想重复我们自己，本书是面向 Elasticsearch 的高级用户的。基础的快照 / 还原 API 我们已经在《Elasticsearch Server, Second Edition》中描述过了，在 Elasticsearch 官方文档中也有提及。现在，我们想聚焦在 Elasticsearch 1.0 版发布之后加入的且在前一本书中被忽略了的功能让我们来讨论 Elasticsearch 的云备份能力。

在云中保存备份

快照 / 还原功能的核心概念是一个仓库。它是一个我们的数据（索引和相关源数据信息）可以安全存储（假设这个仓库是可靠的、高可用的）的地点。假设集群中的每个节点都能够访问仓库，能够读取和写入。由于高可用性和可靠性的需要，Elasticsearch 在附加插件的帮助下，允许我们将数据推送到集群外部，例如云端。通过官方支持的插件，至少有 3 种可以部署的仓库。

- ❑ S3 仓库：亚马逊 Web 服务

- ❑ HDFS 仓库：Hadoop 集群

- ❑ Azure 仓库：微软云平台

因为我们没讨论过任何有关快照 / 还原功能的插件，让我们了解一下它们，看看我们能够将要备份的数据推送到哪里。

1. S3 仓库

S3 仓库是 Elasticsearch 的 AWS 插件的一部分，所以为了使用 S3 作为保存快照的仓库，我们需要首先安装这个插件：

```
bin/plugin -install elasticsearch/elasticsearch-cloud-aws/2.4.0
```

在集群中的每个节点上都安装了插件后，我们需要修改它们的配置（Elasticsearch.yml 文件）来提供 AWS 的访问信息。作为例子的配置如下：

```
cloud:
  aws:
    access_key: YOUR_ACCESS_KEY
    secret_key: YOUT_SECRET_KEY
```

为了创建 Elasticsearch 用来保存快照的 S3 仓库，我们需要执行一个类似下面代码的命令：


```
curl -XPUT 'http://localhost:9200/_snapshot/s3_repository' -d '{
  "type": "s3",
  "settings": {
    "bucket": "bucket_name"
  }
}'
```

在定义基于 S3 的仓库时支持以下的配置。

- ❑ **bucket** : 指定 Elasticsearch 读写数据时使用亚马逊 S3 的哪个桶 (bucket), 这项是必填项。
- ❑ **region**: 指定前面使用的桶所在的 AWS 区域, 默认是 “US Standard”。
- ❑ **base_path** : Elasticsearch 默认将数据写到根目录。这个参数允许你指定想要写入的目录。
- ❑ **server_side_encryption** : 是否开启加密, 默认关闭。如果想使用 AES256 算法来加密数据, 可以设置为 true。
- ❑ **chunk_size** : 指定数据块的大小, 默认为 100MB。如果快照的大小大于 chunk_size, Elasticsearch 会切分数据为不大于 chunk_size 的多个小数据块。
- ❑ **buffer_size** : 缓冲区的大小, 默认 5MB, 也是最小有效值。如果数据块的大小大于 buffer_size, Elasticsearch 会把数据块切分为 buffer_size 大小的多个段, 然后使用 AWS 的 multipart 接口来发送。
- ❑ **max_retries**: 指定 Elasticsearch 在放弃读取或者保存快照前最多重试的次数。默认为 3 次。

除了以上的属性, 我们还可以设置两个属性, 它们可以覆盖保存在 Elasticsearch.yml 文件中用来连接 S3 的身份信息。这在你想使用多个 S3 仓库时非常便利, 每个仓库都有自己的安全配置。

- ❑ **access_key**: 这项覆盖 elasticsearch.yml 文件里的 cloud.aws.access_key。
- ❑ **secret_key**: 这项覆盖 elasticsearch.yml 文件里的 cloud.aws.secret_key。

2. HDFS 仓库

如果你使用 Hadoop 和 HDFS (<http://wiki.apache.org/hadoop/HDFS>) 系统, 备份 Elasticsearch 数据的一个不错的替代就是保存到 Hadoop 集群中。如同 S3 的例子, 有一个专门的插件来实现这个功能。我们使用以下的命令来安装这个插件:

```
bin/plugin -i elasticsearch/elasticsearch-repository-hdfs/2.0.2
```

注意, 有一个专门支持 2.0 版 Hadoop 的插件。这种情况下, 我们应该在插件名后加上 hadoop2 来进行安装。于是对于 Hadoop2, 我们安装插件的命令看起是下面的样子:

```
bin/plugin -i elasticsearch/elasticsearch-repository-hdfs/2.0.2-hadoop2
```

在 Hadoop 已经安装 Elasticsearch 所在的服务器上时，还有一个轻量级的版本。这个版本不包含 Hadoop 类库。要安装这个轻量级的版本，需要使用以下的命令：

```
bin/plugin -i elasticsearch/elasticsearch-repository-hdfs/2.0.2-light
```

当每个 Elasticsearch 节点上都安装了插件（使用的是哪个版本的插件无所谓）且重新启动了集群后，我们可以使用以下的命令在 Hadoop 集群中建立一个仓库：

```
curl -XPUT 'http://localhost:9200/_snapshot/hdfs_repository' -d '{
  "type": "hdfs"
  "settings": {
    "path": "snapshots"
  }
}'
```

我们能够使用的配置如下。

- uri：指定 HDFS 的地址，可选参数，需要满足 hdfs://HOST:PORT 这样的格式。
- path：关于快照需要被存储的路径的信息，这是必填参数。
- load_default：指定是否读取 Hadoop 配置的默认参数，如果需要禁止读取这些参数，可以设置为 false。
- conf_location：Hadoop 配置文件的名称，默认是 extra-cfg.xml。
- chunk_size：指定 Elasticsearch 切分快照数据时使用的数据块的大小。默认是 10MB。如果你想加快保存快照的速度，可以设置一个更小的块大小和更多的流来向 HDFS 推送数据。
- conf.<key>：key 可以是任意的 Hadoop 参数。通过这个属性配置的值会合并到 Hadoop 配置里。
- concurrent_streams：指定一个单一节点使用多少个并发流来向 HDFS 读写数据，默认为 5 个。

3. Azure 仓库

我们想提及的最后一个仓库是微软的 Azure 云。同亚马逊 S3 类似，我们可以使用一个专门的插件来推送索引和元数据到微软的云服务。我们可以通过以下的命令来安装这个插件：

```
bin/plugin -install elasticsearch/elasticsearch-cloud-azure/2.4.0
```

插件的配置也类似亚马逊 S3 的配置。我们的 Elasticsearch.yml 文件需要包含以下的段落：

```
cloud:
  azure:
    storage_account: YOUR_ACCOUNT
    storage_key: YOUR_SECRET_KEY
```

配置好 Elasticsearch 后，我们需要创建具体的仓库，可以通过以下的命令来创建：

```
curl -XPUT 'http://localhost:9200/_snapshot/azure_repository' -d '{  
  "type": "azure"  
}'
```

Elasticsearch 的 Azure 插件支持以下的配置。

- ❑ **container**：跟使用亚马逊 S3 类似，所有的信息都保存在容器中。这项配置指定微软 Azure 空间的容器名称。
- ❑ **base_path**：这项配置允许我们改变 Elasticsearch 存放数据的路径。Elasticsearch 默认把数据保存在根目录。
- ❑ **chunk_size**：Elasticsearch 使用的数据块的最大值（默认 64m，也是允许的最大值）。在数据需要被切分成更小的数据块时，你可以改变这个值。

7.4 联盟搜索

有时把数据保存在一个集群中是不够的。想象以下情况，你有多个地点需要索引和搜索数据，例如：本地公司部门用他们自己的集群来保存数据。公司的数据中心可能也想搜索这些数据，不是一个地点一个地点地搜索，而是一次搜索全部。当然，在你的搜索应用中，你可以连接全部的这些集群，然后合并这些结果，但是从 Elasticsearch 的 1.0 版开始，还可以使用部落节点（tribe node）来实现。部落节点作为联合客户端可以提供访问多个 Elasticsearch 集群的能力。部落节点的功能是从连接的集群中获取所有的集群状态，并合并这些状态为一个全局的状态。在本小节中，我们会讨论一下部落节点，以及怎么配置和使用它们。



注意

请记住我们讨论的功能是在 Elasticsearch 的 1.0 版之后引入的，并且仍然被标记为实验性的。在将来的版本中它可能被修改甚至被移除。

7.4.1 测试用的集群

为了想你展示部落集群是如何工作的，我们会创建两个保存数据的集群。第 1 个集群叫作 `mastering_one`（想必你还记得怎么设置集群名称，你需要在 `Elasticsearch.yml` 文件中设置 `cluster.name` 属性），第 2 个集群叫作 `mastering_two`。为了尽可能地简单，每个集群都只包含一个节点。集群 `mastering_one` 的唯一节点的 IP 地址是 192.168.56.10，集群 `mastering_two` 的唯一节点的 IP 地址是 192.168.56.40。

集群 1 中索引了以下的文档：

```
curl -XPOST '192.168.56.10:9200/index_one/doc/1' -d '{"name" : "Test document 1 cluster 1"}'
curl -XPOST '192.168.56.10:9200/index_one/doc/2' -d '{"name" : "Test document 2 cluster 1"}'
```

集群 2 中索引了以下的文档：

```
curl -XPOST '192.168.56.40:9200/index_two/doc/1' -d '{"name" : "Test document 1 cluster 2"}'
curl -XPOST '192.168.56.40:9200/index_two/doc/2' -d '{"name" : "Test document 2 cluster 2"}'
```

7.4.2 建立部落节点

现在，让我们试着创建一个简单的部落节点，默认使用多播发现。为此，我们需要一个新的 Elasticsearch 节点。我们同样需要为这个节点提供一个配置来指定部落节点需要连接的集群，在我们的例子里，就是之前创建的两个集群。为了配置我们的部落节点，我们需要 Elasticsearch.yml 文件中存在以下内容：

```
tribe.mastering_one.cluster.name: mastering_one
tribe.mastering_two.cluster.name: mastering_two
```

部落节点的配置项都以 tribe 为前缀。在上面的配置中，我们告诉 Elasticsearch 我们有两个部落，一个叫作 mastering_one，一个叫作 mastering_two。部落名可以是任意的名称，只是用来区别不同的集群。

我们可以启动部落节点了，我们将在 IP 地址是 192.156.56.50 的服务器上启动它。启动后，我们将尝试使用默认的多播发现来找到 mastering_one 和 mastering_two 集群，并连接它们。你会在部落节点的 log 中看到下面的内容：

```
[2014-10-30 17:28:04,377][INFO ][cluster.service
]
[Feron] added {[mastering_one_node_1] [mGF6HHoORQGYkVTzuPd4Jw]
[ragnar] [inet[/192.168.56.10:9300]]{tribe.name=mastering_one}},,
reason: cluster event from mastering_one, zen-disco-receive(from
master [[mastering_one_node_1] [mGF6HHoORQGYkVTzuPd4Jw] [ragnar]
[inet[/192.168.56.10:9300]]])
[2014-10-30 17:28:08,288][INFO ][cluster.service
]
[Feron] added {[mastering_two_node_1] [ZqvDAsY1RmylH46hqCTEnw]
[ragnar] [inet[/192.168.56.40:9300]]{tribe.name=mastering_two}},,
reason: cluster event from mastering_two, zen-disco-receive(from
master [[mastering_two_node_1] [ZqvDAsY1RmylH46hqCTEnw] [ragnar]
[inet[/192.168.56.40:9300]]])
```

可以看到，部落节点将两个集群联合到了一起。

使用单播发现来组成部落

当然了，使用多播发现来组建部落不是唯一的办法，如果需要，我们也可以使用单播

发现。例如，想让部落节点使用单播模式，我们可以把 Elasticsearch.yml 文件修改为以下的样子：

```
tribe.mastering_one.cluster.name: mastering_one
tribe.mastering_one.discovery.zen.ping.multicast.enabled: false
tribe.mastering_one.discovery.zen.ping.unicast.hosts:
["192.168.56.10:9300"]
tribe.mastering_two.cluster.name: mastering_two
tribe.mastering_two.discovery.zen.ping.multicast.enabled: false
tribe.mastering_two.discovery.zen.ping.unicast.hosts:
["192.168.56.40:9300"]
```

正如你所看到的，对于每一个部落集群，我们禁止了多播，并指定了单播的主机地址。再提一下我们已经写过的，部落节点的每个属性都是以 tribe 为前缀的。

7.4.3 通过部落节点读取数据

我们前面讲过，部落节点从所有连接的集群中获取集群状态，并合并为一个集群状态。这样做是为了在使用部落节点进行读写操作时，让这些操作在所有的集群上执行。由于集群状态合并过了，几乎所有的操作的工作原理都跟它们在单一集群中执行时一样，例如搜索。

让我们试着在部落节点上运行一个单一的查询，看看我们能得到什么。为此，我们使用如下的命令：

```
curl -XGET '192.168.56.50:9200/_search?pretty'
```

以上查询的结果如下：

```
{
  "took" : 9,
  "timed_out" : false,
  "_shards" : {
    "total" : 10,
    "successful" : 10,
    "failed" : 0
  },
  "hits" : {
    "total" : 4,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "index_two",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 1.0,
      "_source": {"name" : "Test document 1 cluster 2"}
    }, {
      "_index" : "index_one",
      "_type" : "doc",
      "_id" : "2",
      "_score" : 1.0,
```

```

    "_source":{"name" : "Test document 2 cluster 1"}
  }, {
    "_index" : "index_two",
    "_type" : "doc",
    "_id" : "2",
    "_score" : 1.0,
    "_source":{"name" : "Test document 2 cluster 2"}
  }, {
    "_index" : "index_one",
    "_type" : "doc",
    "_id" : "1",
    "_score" : 1.0,
    "_source":{"name" : "Test document 1 cluster 1"}
  } ]
}
}

```

正如你所看到的，我们得到了来自两个集群的文档。我们的部落节点从所有连接的部落中得到数据，然后返回相关的结果。当然了，我们可以执行更加复杂的查询，我们可以使用过滤（percolation）、联想（suggestions）等。

主节点级别的读操作

需要主节点存在的读操作会在部落集群中执行，例如读取集群状态或者集群健康情况。例如，我们看下部落节点返回的集群健康数据是怎样的。我们可以用如下的命令来查看：

```
curl -XGET '192.168.56.50:9200/_cluster/health?pretty'
```

以上命令的结果与如下的内容类似：

```

{
  "cluster_name" : "elasticsearch",
  "status" : "yellow",
  "timed_out" : false,
  "number_of_nodes" : 5,
  "number_of_data_nodes" : 2,
  "active_primary_shards" : 10,
  "active_shards" : 10,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 10
}

```

正如你所看到的，部落节点报告存在 5 个节点。每个集群有 1 个节点，一个部落节点和部落节点内的两个用来连接集群的内部节点。这是为什么有 5 个节点而不是 3 个的原因。

7.4.4 通过部落节点写入数据

我们讨论过了查询和主节点级别的读操作，现在是时候使用部落节点向 Elasticsearch

写入一些数据了。我们不会过多讨论索引过程，而是仅仅尝试向一个我们已经连接的集群中索引一些额外的文档。我们可以执行如下的命令来完成这项工作：

```
curl -XPOST '192.168.56.50:9200/index_one/doc/3' -d '{"name" : "Test document 3 cluster 1"}'
```

执行上面的命令会得到如下的响应：

```
{ "_index": "index_one", "_type": "doc", "_id": "3", "_version": 1, "created": true }
```

正如你所看到的，文档被创建了，并且被索引在了正确的集群上。部落节点只是将请求在内部转发给正确的集群。所有不要求改变集群状态的写操作，例如索引，都能通过部落节点正确地执行。

主节点级别的写操作

主节点级别的写操作不能在部落节点上执行。例如，我们不能通过部落节点创建索引。创建索引这类操作在部落节点上执行时会失败，因为没有全局的主节点存在。通过运行如下的命令我们可以很容易地验证这个结论：

```
curl -XPOST '192.168.56.50:9200/index_three'
```

以上的命令会在等待大约 30 秒后返回如下的错误信息：

```
{ "error": "MasterNotDiscoveredException[waited for [30s]]", "status": 503 }
```

正如你所看到的，索引没有创建。我们应当在组成部落的集群上来运行主节点级别的写操作。

7.4.5 处理索引冲突

部落节点不能正确处理的事情之一就是在其连接的多个集群中有同名的索引。Elasticsearch 的部落节点的默认行为是从中只选择一个。所以，如果你的多个集群中有相同的索引，只有一个会被选择。

为了验证这个特性，我们在 `mastering_one` 集群和 `mastering_two` 集群上创建名为 `test_conflicts` 索引。我们可以使用如下的命令来创建：

```
curl -XPOST '192.168.56.10:9200/test_conflicts'
curl -XPOST '192.168.56.40:9200/test_conflicts'
```

除此之外，我们再索引两个文档，每个集群一个。我们使用了如下的命令：

```
curl -XPOST '192.168.56.10:9200/test_conflicts/doc/11' -d '{"name" : "Test conflict cluster 1"}'
curl -XPOST '192.168.56.40:9201/test_conflicts/doc/21' -d '{"name" : "Test conflict cluster 2"}'
```

现在我们在部落节点上执行一个简单的查询命令：

```
curl -XGET '192.168.56.50:9202/test_conflicts/_search?pretty'
```

命令的输出如下：

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "test_conflicts",
      "_type" : "doc",
      "_id" : "11",
      "_score" : 1.0,
      "_source": {"name" : "Test conflict cluster 1"}
    } ]
  }
}
```

如你所见，我们的结果中只包含一个文档。这是由于 Elasticsearch 部落节点不能处理来自不同集群的同名索引，只会选择一个。这非常危险，因为我们不知道能期望得到什么。

一个好的事情是我们可以通过在 Elasticsearch.yml 中指定 `tribe.on_conflict`（在 Elasticsearch 1.2.0 时引入）属性来控制这个行为。我们可以配置为以下值之一。

- ❑ `any`：这项是 Elasticsearch 的默认值。Elasticsearch 会从连接的部落集群中选择一个索引。
- ❑ `drop`：Elasticsearch 会忽略同名索引，并排除在全局集群状态之外。这意味着在使用部落节点时这些索引对读写都是不可见的，但仍然会存在于连接到部落节点的集群上。
- ❑ `prefer_TRIBE_NAME`：Elasticsearch 允许我们选择哪个集群的索引。例如，如果我们设置为 `prefer_mastering_one`，这意味着 Elasticsearch 会从冲突的索引中选择 `mastering_one` 集群上的那个索引。

7.4.6 屏蔽写操作

部落节点也可以配置为屏蔽所有的写操作和所有的修改元数据的请求。为屏蔽所有的写请求，我们需要设置 `tribe.blocks.write` 属性为 `true`。为禁止元数据修改请求，我们需要设置 `tribe.blocks.metadata` 属性为 `true`。这两个属性默认为 `false`，这意味着允许写操作和更改

元数据请求。当部落节点应该仅被用来执行搜索时就可以禁用这些操作了。

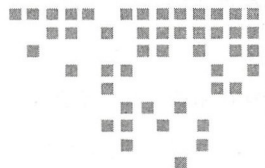
另外，Elasticsearch 1.2.0 引入了在指定索引上屏蔽写操作的能力。我们通过设置 `tribe.blocks.indices.write` 属性为需要屏蔽的索引名来实现这个功能。例如，如果我们希望部落节点屏蔽所有以 `test` 和 `production` 开头的索引上的写操作，可在 `elasticsearch.yml` 文件中做如下的配置：

```
tribe.blocks.indices.write: test*, production*
```

7.5 小结

在本章中，我们更多地关注于 Elasticsearch 的配置和其在 1.0 版之后引入的新特性。我们配置了发现和恢复模块，使用了人类友好的 Cat API。另外我们使用了备份和还原功能，这个功能可以方便地备份和还原索引。最后，我们着眼于联盟搜索，以及如何在多个集群上搜索和索引数据。还是使用 Elasticsearch 提供的功能，且只需要连接到一个节点就能实现。

在下一章中，我们会关注于 Elasticsearch 的性能方面。我们会从使用过滤器（Filter）优化查询开始。我们会讨论垃圾回收器的工作，我们会使用 Elasticsearch 新的基准测试（benchmark）功能来对我们的查询进行基准测试。我们会使用预热查询来减少查询的执行时间，会使用热点线程 API 来查看 Elasticsearch 里正在发生什么。最后，我们会讨论 Elasticsearch 调优和在高索引和查询压力场景下如何调整 Elasticsearch。



第 8 章

Chapter 8

提高性能

在上一章中，我们探讨了发现和恢复模块的配置。我们配置了这些模块，知道了它们为什么是重要的。我们还关注了一些通过插件实现的可用发现模块，使用了对人类友好的 Cat API，以人类可读的格式来获取集群信息，备份数据到外部的云存储，还一起探索了部落节点（一个联盟搜索功能，它允许我们把一些 Elasticsearch 集群连接到一起）。到本章的结束时，将涵盖以下内容：

- ❑ 在使用基于字段缓存的查询时，doc values 能给我们什么帮助
- ❑ 垃圾回收器是如何工作的
- ❑ 在上线前如何对查询做基准测试并修复性能问题
- ❑ 什么是热点线程 API 以及它怎样帮你调试问题
- ❑ 如何优化 Elasticsearch 以及在优化时要观察什么
- ❑ 在高查询吞吐量场景下优化 Elasticsearch
- ❑ 在高索引吞吐量场景下优化 Elasticsearch

8.1 使用doc values来优化查询

在 6.6 节中，我们描述了缓存：用来提高 Elasticsearch 的杰出性能的方法之一。不幸的是，缓存不是万能的，有时不使用缓存会更好。如果你的数据频繁更新，并且查询具有唯一性且不可重复，那么缓存不会真的帮到你什么，甚至有时会让性能变得更加糟糕。

8.1.1 字段缓存存在的问题

每个缓存都基于几个简单的原理。主要的设想是：为了提高性能，避免从较慢的类似机械硬盘的数据源获取数据，或者减少系统重新计算数据的需求，保存一部分数据到内存中是值得的。然而，缓存不是免费的，有着它的代价。对于 Elasticsearch 来说，缓存的代价主要是内存。根据缓存类型的不同，你可能只需要保存最近使用的数据，但是同样，这并不总是能够实现的。有时，容纳全部的信息是必须的，因为不然的话，缓存就毫无意义。例如，用来在排序和聚合时使用的字段缓存，为了让缓存有作用，指定字段的所有值都必须被 Elasticsearch 实例 uninvverted，并保存在缓存里。如果我们有大量的文档，并且分片也非常大，那么可能会遇到麻烦。这类麻烦的征兆有时会使 Elasticsearch 对于查询返回如下响应：

```
{
  "error": "ReduceSearchPhaseException[Failed to execute phase
[fetch], [reduce] ; shardFailures {[vWD3FNVoTy-
64r2vf6NwAw] [dvt1] [1]: ElasticsearchException[Java heap space];
nested: OutOfMemoryError[Java heap space]; }{[vWD3FNVoTy-
64r2vf6NwAw] [dvt1] [2]: ElasticsearchException[Java heap space];
nested: OutOfMemoryError[Java heap space]; }]; nested:
OutOfMemoryError[Java heap space]; ",
  "status": 500
}
```

另一个跟内存有关的问题的标志可能存在于 Elasticsearch 日志中，看起来是下面的样子：

```
[2014-11-29 23:21:32,991] [DEBUG] [action.search.type      ]
[Abigail Brand] [dvt1] [2], node[vWD3FNVoTy-64r2vf6NwAw], [P],
s[STARTED]: Failed to execute
[org.elasticsearch.action.search.SearchRequest@49d609d3]
lastShard [true]
org.elasticsearch.ElasticsearchException: Java heap space
  at org.elasticsearch.ExceptionsHelper.convertToRuntime
    (ExceptionsHelper.java:46)
  at org.elasticsearch.search.SearchService.executeQueryPhase
    (SearchService.java:304)
  at org.elasticsearch.search.action.
    SearchServiceTransportAction$5.call
    (SearchServiceTransportAction.java:231)
  at org.elasticsearch.search.action.
    SearchServiceTransportAction$5.call
    (SearchServiceTransportAction.java:228)
  at org.elasticsearch.search.action.
    SearchServiceTransportAction$23.run
    (SearchServiceTransportAction.java:559)
  at java.util.concurrent.ThreadPoolExecutor.runWorker
    (ThreadPoolExecutor.java:1145)
  at java.util.concurrent.ThreadPoolExecutor$Worker.run
    (ThreadPoolExecutor.java:615)
  at java.lang.Thread.run(Thread.java:744)
Caused by: java.lang.OutOfMemoryError: Java heap space
```

这就是 doc values 可以帮助我们的情形。doc values 是 Lucene 中基于列的数据结构，

这意味着它们不将数据保存在倒排索引中，而是保存在一个基于文档的数据结构里，并存储在磁盘上，在索引文档时就计算好。于是，doc values 使得我们避免在字段缓存中保存 unindexed 的数据，代替的是 doc values 从索引中获取数据。从 Elasticsearch 1.4.0 开始，doc values 的访问与使用字段缓存一样快。

8.1.2 使用 doc values 的例子

为了向你展示基于 doc values 和基于字段数据缓存在内存消耗上的不同，我们索引一些简单的文档到 Elasticsearch 中。我们索引相同的数据到两个索引中：dvt1 和 dvt2。它们的结构相同，唯一的下面是下面代码中的粗体部分：

```
{
  "t": {
    "properties": {
      "token": {
        "type": "string",
        "index": "not_analyzed",
        "doc_values": true
      }
    }
  }
}
```

索引 dvt2 使用 doc values，而 dvt1 不使用，因此 dvt1 上查询（如果使用了排序和聚合）会使用字段数据缓存。



注意 为了便于测试，我们设置 JVM 堆的大小低于 Elasticsearch 的默认值。Elasticsearch 实例化启动时使用：

```
bin/elasticsearch -Xmx16m -Xms16m
```

这起初看起来有些荒谬，但是谁说我们不能在嵌入式设备上使用 Elasticsearch 呢？当然了，另一个模拟这个问题的办法是索引更多的数据。然而，只是为了测试的话，保持小内存完全足够了。

现在来看看当命中我们的示例索引时 Elasticsearch 的行为是怎样的。查询看起来并不复杂，但能够很好地展现问题。我们尝试基于文档的一个字段来排序数据：token。我们知道，排序需要 unindexed 的数据，所以它要么使用字段数据缓存，要么在 DocValues 存在时使用 doc values。查询本身看起来是这样的：

```
{
  "sort": [
    {
      "token": {
```



```

        "order": "desc"
    }
}
]
}

```

这是一个简单的排序，但是在我们尝试搜索 dvt1 索引时已经足够让我们的服务器宕机了。同时，在 dvt2 索引上执行的查询返回了期望的结果，没有任何出问题的迹象。

内存使用上的差别非常明显。在启动参数中删除内存限制，并重新启动 Elasticsearch 后，我们可以对比两个索引的内存使用情况。在 dvt1 和 dvt2 两个索引上都执行了查询后，可使用如下的命令来查看内存使用情况：

```
curl -XGET 'localhost:9200/dvt1,dvt2/_stats/fielddata?pretty'
```

在我们的例子中，Elasticsearch 的响应如下：

```

{
  "_shards" : {
    "total" : 20,
    "successful" : 10,
    "failed" : 0
  },
  "_all" : {
    "primaries" : {
      "fielddata" : {
        "memory_size_in_bytes" : 17321304,
        "evictions" : 0
      }
    },
    "total" : {
      "fielddata" : {
        "memory_size_in_bytes" : 17321304,
        "evictions" : 0
      }
    }
  },
  "indices" : {
    "dvt2" : {
      "primaries" : {
        "fielddata" : {
          "memory_size_in_bytes" : 0,
          "evictions" : 0
        }
      },
      "total" : {
        "fielddata" : {
          "memory_size_in_bytes" : 0,
          "evictions" : 0
        }
      }
    }
  },
}

```

```

    "dvt1" : {
      "primaries" : {
        "fielddata" : {
          "memory_size_in_bytes" : 17321304,
          "evictions" : 0
        }
      },
      "total" : {
        "fielddata" : {
          "memory_size_in_bytes" : 17321304,
          "evictions" : 0
        }
      }
    }
  }
}

```

最有趣的部分已经被加粗显示了。你能看到，没有使用 doc values 的索引使用了 17321304 个字节（16MB）的内存来保存字段数据缓存。与此同时，第 2 个索引完全没有使用任何 RAM 内存来保存 uninverted 数据。

当然，同大多数优化类似，对于资源来说使用 doc values 并不是免费的。使用 doc values 的缺点之一是速度，doc values 要比字段数据缓存慢。另一个缺点是需要额外的空间来保存 doc values。例如，在我们简单的测试例子里，使用了 DocValues 的索引大小是 41MB，而没有使用 doc values 的是 34MB。这给索引大小带来了大约 20% 以上的增长，但是这个通常取决于索引中的数据。然而，记得当遇到与查询和字段数据缓存有关的内存问题时，你可能会想要开启 doc values，重新索引你的数据，然后就再也不用担心与字段数据缓存相关的内存出现溢出异常了。

8.2 了解垃圾回收器

Elasticsearch 是一个 Java 应用，因此，它在 Java 虚拟机中运行。每个 Java 应用都会被编译为叫作字节码的东西，它可以被 JVM 执行。用最一般的思考方式，你可以想象 JVM 只是执行其他程序并控制它们的行为。然而，除非你为 Elasticsearch 开发了插件（会在第 9 章中讨论插件），否则，你并不需要考虑这个。你需要关心的垃圾回收器，它是 JVM 中负责内存管理的部分。当对象不再被引用时，它们可以被垃圾回收器从内存中移除。当内存运行时，低优先级的垃圾回收器开始工作，尝试回收那些不再被引用的对象。在本节中，我们会看到如何配置垃圾回收器，如何避免内存交换，如何记录垃圾回收器的行为，怎样调试异常，怎样使用 Java 工具查看垃圾回收器是如何工作的。



你可以从互联网上学习更多关于 JVM 架构的信息，例如，维基百科：http://en.wikipedia.org/wiki/Java_virtual_machine。

8.2.1 Java 内存

当我们使用 Xms 和 Xmx 参数（或者 ES_MIN_MEM 和 ES_MAX_MEM 属性）来指定内存时，我们指定了最小和最大的 JVM 堆。它基本上是可以被 Java 程序（对于我们来说就是 Elasticsearch）使用的物理内存上的保留区。一个 Java 进程从不会使用比我们通过 Xmx 参数（或者 ES_MAX_MEM 属性）指定的多的堆内存。当一个对象在一个 Java 应用中被创建，它就被放置在堆内存上。当它不再被使用后，垃圾回收器会尝试从堆上回收它来释放内存空间，以使 JVM 能够在将来重用这个空间。可以想象如果没有足够的堆内存来供你的应用在堆上创建新对象，那么不好的事情就会发生了。JVM 会抛出一个 OutOfMemory 异常，这是一个内存出了问题的迹象，要么是没有足够的内存给它，要么是有内存泄漏，导致没有释放不再使用的对象。



当在性能足够强悍且有大量剩余内存的机器上运行 Elasticsearch 时，可能会问自己，是运行一个分配了大量内存给 JVM 的 Elasticsearch 大型实例好，还是运行一些堆内存小些的实例好。在回答这个问题之前，我们需要记得分配的 JVM 的堆内存越多，垃圾回收器的工作就越困难。并且，当设置堆的大小超过 32GB 时，我们不能从指针压缩中获得好处，JVM 将为数据使用 64 位的指针，这意味着我们将使用更多的内存来保存相同数量数据的地址。基于这些因素，通常使用多个小些的 Elasticsearch 实例比一个大实例更好些。

在 Java 7 中 JVM 内存被分成如下的区域。

- ❑ eden space：这是堆内存中 JVM 最初分配大多数类型的对象的部分。
- ❑ survivor space：这是保存从 eden 空间的垃圾回收中幸存下来的对象的部分。survivor 空间被分成 survivor0 和 survivor1。
- ❑ tenured generation：这里是容纳在 survivor 空间里生存了一段时间的对象的部分。
- ❑ permanent generation：这里是非堆内存，存储虚拟机自身数据的地方，例如类和对象的方法都保存在这里。
- ❑ code cache：这里是非堆内存，它存于 HostSpot JVM 中，用来编译和存储 native 代码。

前面的分类可以简化。eden 空间和 survivor 空间被叫作年轻代堆空间，tenured generation 通常被称为老年代。

Java 对象的生命周期和垃圾回收

为了明白垃圾回收器是如何工作的，让我们详细了解一下 Java 对象的生命周期。

当一个对象在 Java 应用中被创建后，它被放置在年轻代的 eden 空间中。然后，当下一次年轻代垃圾回收运行，且这个对象从中幸存下来（基本上，如果它不是一个一次性使用的对象，应用仍然在使用它）时，它会被移动到年轻代的 survivor 空间中（开始是 survivor0，然后经过下一次年轻代垃圾回收，移动到 survivor1）。

在 survivor1 空间中存活了一段时间后，对象被移动到 tenured generation 空间，于是它现在是老年代的一部分了。从现在开始，年轻代垃圾回收器不能够在堆空间中移动这个对象了。现在，这个对象会存活在老年代中直到我们的应用决定不再需要它了。在这种情形下，当下一次 full gc 到来时，它会从堆空间中移除，为新的对象留出空间。



注意

通常你需要努力实现的是小而多次的垃圾回收，而不是一次长时间的回收。这是因为你希望你的应用以稳定的性能水平运行，且垃圾回收器的工作对 Elasticsearch 是透明的。当一个大型的垃圾回收发生时，它可能会是一个 stop the world 类型的垃圾回收事件，这时 Elasticsearch 会冻结一小段时间，这会让查询变得缓慢，且索引过程会停止一段时间。

基于前面的信息，我们可以说（也确实如此）至少到目前为止，Java 使用分代垃圾回收：对象从垃圾回收中幸存下来的次数越多，就有越大的机会晋升。于是，可以说有两种类型的垃圾回收器同时存在：年轻代垃圾回收器（也被称为 minor）和老年代垃圾回收器（也被称为 major）。



注意

随着 Java 7 update 9 的释出，Oracle 引入了一个新的垃圾回收器，叫做 G1。它承诺完全不受 stop the world 事件的影响，并且应该比其他垃圾回收器更快。想阅读更多关于 G1 的信息，请查看 <http://www.oracle.com/technetwork/tutorials/tutorials-1876574.html>。尽管 Elasticsearch 创立者们不建议使用 G1，但众多的公司成功地使用了它，并使得他们在大数据量和重度查询的环境中使用 Elasticsearch 时克服了 stop the world 事件带来的问题。

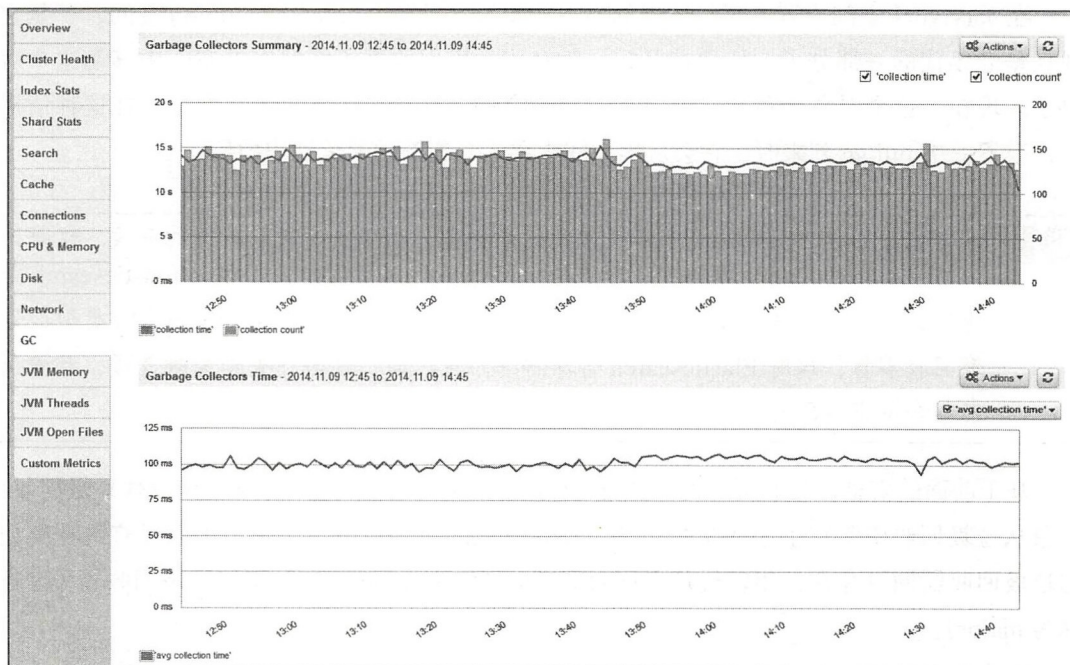
8.2.2 解决垃圾回收问题

在解决垃圾回收问题时，你需要识别的第一件事是问题的源头。这不是一件简单的工作，通常需要系统管理员或者负责管理集群的人员付出一些努力。在本节中，我们会向你展示两种方法来观察和识别垃圾回收器的问题。第 1 个是打开 Elasticsearch 的垃圾回收日

志，第 2 个是使用 jstat 命令，它在大多数的 Java 分发包中存在。

除了以上的方法，请记住还有其他工具能帮助你调试与内存和垃圾回收器相关的问题。这些工具通常以软件监控解决方案的形式提供，例如 Sematext Group SPM (<http://sematext.com/spm/index.html>) 或者 NewRelic (<http://newrelic.com/>)。这些解决方案提供了详细的信息，不仅是关于垃圾回收的，还有整体内存使用的。

一个来自前面提到的 SPM 应用的样例面板展示了垃圾回收器的工作情况：



打开垃圾回收器的工作日志

Elasticsearch 允许我们观察垃圾回收器超长时工作的周期。在默认的 Elasticsearch.yml 文件中，你能看到如下的项，其默认被注释了：

```
monitor.jvm.gc.young.warn: 1000ms
monitor.jvm.gc.young.info: 700ms
monitor.jvm.gc.young.debug: 400ms
monitor.jvm.gc.old.warn: 10s
monitor.jvm.gc.old.info: 5s
monitor.jvm.gc.old.debug: 2s
```

如同你看到的那样，配置指定了 3 个 log 等级和每个的阈值。例如，对于 info 日志等级，如果年轻代回收花费了 700 毫秒或者更多，Elasticsearch 会把信息写入日志。对于老年代，如果花费超过 5s 则会被计入日志。



注意

在老版本的 Elasticsearch 中（1.0 之前），记录年轻代垃圾回收信息的前缀是 `monitor.jvm.gc.ParNew.*`，而记录老年代垃圾回收信息的前缀是 `monitor.jvm.gc.ConcurrentMarkSweep.*`。

你在日志中看到的内容会是下面这样：

```
[2014-11-09 15:22:52,355][WARN ][monitor.jvm
[Lizard] [gc][old][964][1] duration [14.8s], collections
[1]/[15.8s], total [14.8s]/[14.8s], memory [8.6gb]-
>[3.4gb]/[11.9gb], all_pools {[Code Cache] [8.3mb]-
>[8.3mb]/[48mb]}{[young] [13.3mb]->[3.2mb]/[266.2mb]}{[survivor]
[29.5mb]->[0b]/[33.2mb]}{[old] [8.5gb]->[3.4gb]/[11.6gb]}
```

如同你看到的，log 文件的首行告诉我们这是关于老年代垃圾回收器的工作。我们能够看到总的回收时间是 14.8 秒。在进行垃圾回收之前，使用了 8.6GB 的堆内存（总的堆内存是 11.9GB）。垃圾回收后，堆内存的使用量被缩减到 3.4GB。这之后，你能看到关于堆的哪个部分被垃圾回收器所涵盖的详细信息：代码缓存、年轻代空间、survivor 空间和老年代堆空间。

当以特定的阈值来开启垃圾回收器的工作日志时，通过查看日志能够发现事情在何时不再以所希望的方式运行。然而，如果想看到更多，Java 提供了一个工具：jstat。

1. 使用 JStat

运行 jstat 命令来查看垃圾回收器如何工作的操作非常简单，执行如下的命令即可：

```
jstat -gcutil 123456 2000 1000
```

开关 -gcutil 表示监控垃圾回收器的工作，123456 是运行着 Elasticsearch 的虚拟机的标识符，2000 是以毫秒表示的采样周期，1000 是采样的数量。所以，对于我们的例子，前面的命令会在 33 分钟（ $2000 \times 1000 / 1000 / 60$ ）多一点的时间后返回。

在大多数的情况下，虚拟机标识符同进程 ID 类似，甚至相同，但并不总是这样。为了查看哪个 Java 进程在运行和它们的虚拟机标识符是什么，可以执行 jps 命令，绝大多数的 Java 分发包都提供。一个执行它的例子如下：

```
jps
```

结果如下：

```
16232 Jps
11684 ElasticSearch
```

在 jps 命令的结果中，我们看到每一行都包含了 JVM 的标识符，后面跟着的是进程的名称。如果想学习更多关于 jps 命令的信息，请查看 Java 文档，地址是：<http://docs.oracle.com/javase/7/docs/technotes/tools/share/jps.html>。



注意

请记得使用与运行 Elasticsearch 相同的账号来运行 `jstat` 命令，如果不能的话，使用管理员权限来运行（例如在 Linux 系统上使用 `sudo` 命令）。拥有访问 Elasticsearch 所在进程的权限是至关重要的，否则 `jstat` 命令就不能够连接到那个进程。

现在，我们来看一个 `jstat` 命令输出的例子：

S0	S1	E	O	P	YGC	YGCT	FGC	FGCT	GCT
12.44	0.00	27.20	9.49	96.70	78	0.176	5	0.495	0.672
12.44	0.00	62.16	9.49	96.70	78	0.176	5	0.495	0.672
12.44	0.00	83.97	9.49	96.70	78	0.176	5	0.495	0.672
0.00	7.74	0.00	9.51	96.70	79	0.177	5	0.495	0.673
0.00	7.74	23.37	9.51	96.70	79	0.177	5	0.495	0.673
0.00	7.74	43.82	9.51	96.70	79	0.177	5	0.495	0.673
0.00	7.74	58.11	9.51	96.71	79	0.177	5	0.495	0.673

上面的例子来自 Java 文档，我们决定采用它是由于它很好地向我们呈现了 `jstat` 是关于什么的。现从说明每列的含义开始。

- ❑ S0: survivor0 空间的使用情况，以空间容量的百分比表示。
- ❑ S1: survivor1 空间的使用情况，以空间容量的百分比表示。
- ❑ E: eden 空间的使用情况，以空间容量的百分比表示。
- ❑ O: 老年代空间的使用情况，以空间容量的百分比表示。
- ❑ YGC: 年轻代垃圾回收的次数。
- ❑ YGCT: 年轻代垃圾回收消耗的时间。
- ❑ FGC: full gc 的次数。
- ❑ FGCT: full gc 消耗的时间。
- ❑ GCT: gc 的总用时。

现在，回到例子中。你可以看到，在第 3 个样本后和第 4 个样本前有一次年轻代垃圾回收事件。我们能看出这次回收耗时 0.001s（第 4 个样本的 YGCT 值 0.117 减去第三个样本的 YGCT 值 0.116）。我们也知道这次回收从 eden 空间（第 4 个样本中的使用百分比是 0，而第 3 个样本中的使用百分比是 83.97）晋升对象到老年代堆空间（从第 3 个样本的 9.49% 增长到了第 4 个样本的 9.51%）。这个例子向你展示了如何分析 `jstat` 的输出。当然，这会耗用一定的时间，并且需要一些关于垃圾回收器如何工作的知识，以及堆上都保存了什么。然而，有时，这是分析为什么 Elasticsearch 在特定时刻卡住的唯一方式。

记住，如果你见到 Elasticsearch 没有正常工作，S0、S1 或者 E 列显示 100%，并且垃圾回收器不能回收这些堆空间，那么，要么是年轻代太小了需要增加（当然，如果有充足的剩余物理内存），要么是内存问题。这些问题可能与内存泄漏有关，即一些资源没有释放不

再使用的内存。另一方面，当你的老年代空间达到 100% 时，垃圾回收器努力回收它（频繁的垃圾回收），但却不能回收，那么这大概意味着没有足够的堆空间来让 Elasticsearch 节点正常工作。这时，在不改变索引结构的前提下，你能做的是增加运行 Elasticsearch 的 JVM 的可用堆空间（关于 JVM 参数的更多信息，参见 <http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html>）。

2. 创建内存 dump

我们到目前为止都还没讨论的一件事是 dump 堆内存到一个文件的能力。Java 允许我们获得一个给定时点的内存快照，我们可以使用这个快照来分析内存中存储了什么以及发现问题。为了 dump 出 Java 进程内存，可以使用 jmap（<http://docs.oracle.com/javase/7/docs/technotes/tools/share/jmap.html>）命令，例如，像这样：

```
jmap -dump:file=heap.dump 123456
```

在我们的例子里，“123456”是想要获得内存 dump 的 Java 进程的标识符，“-dump:file=heap.dump”指定我们想要保存 dump 到一个名称为 heap.dump 的文件里。通过使用特殊软件，这个 dump 可以被进一步分析，例如使用 jhat（<http://docs.oracle.com/javase/7/docs/technotes/tools/share/jhat.html>），但是如何使用这些工具超出了本书的讨论范围。

3. 关于垃圾回收器工作的更多信息

优化垃圾回收不是一个简单的过程。使用 Elasticsearch 为我们设置的默认选项进行部署，这在绝大多数情况下通常是足够的，你需要做的唯一一件事是调整节点的内存数量。调优垃圾回收器工作的主题超出了本书的范畴，它的范围广阔，被一些开发者称为“黑魔法”。然而，如果想阅读更多关于垃圾回收器的问题，如它们有哪些选项，它们如何影响你的应用，这里推荐一篇极好的文章，你可在：<http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html> 上找到。尽管这篇文章是关于 Java 6 的，但是绝大多数的选项都能使用在 Java 7 的部署上。

4. 调整 Elasticsearch 的垃圾回收器的工作

现在我们知道了垃圾回收器是如何工作的，以及怎样调试它的问题，所以知道如何调整 Elasticsearch 启动参数来改变垃圾回收器的工作是很有益的。这取决于是如何运行 Elasticsearch 的。我们会着眼于两个最常用的方式：Elasticsearch 的分发包提供的标准的启动脚本和使用服务包装器（service wrapper）。

5. 使用标准启动脚本

当使用标准启动脚本来增加额外的 JVM 参数时，我们应该把它们包含进 JAVA_OPTS 环境变量中。例如，对于类似 Linux 的系统，如果我们想包含 -XX:+UseParNewGC

-XX:+UseConcMarkSweepGC 到 Elasticsearch 的启动参数中，我们会做如下的事情：

```
export JAVA_OPTS="-XX:+UseParNewGC -XX:+UseConcMarkSweepGC"
```

为了查看属性是否配置成功了，我们只需执行另一个命令：

```
echo $JAVA_OPTS
```

前面的命令对于我们的例子应该返回以下的输出：

```
-XX:+UseParNewGC -XX:+UseConcMarkSweepGC
```

6. 服务包装器

Elasticsearch 允许我们使用服务包装器来把它安装为一个服务（<https://github.com/Elasticsearch/Elasticsearch-servicewrapper>）。如果你正在使用服务包装器，设置 JVM 参数的方法就与前面展示的方法不同了。我们需要做的是修改 Elasticsearch.conf 文件，它大概会被放在 /opt/Elasticsearch/bin/service/（如果你的 Elasticsearch 安装在 /opt/Elasticsearch）。在 Elasticsearch.conf 文件中，你会看到一些属性，例如：

```
set.default.ES_HEAP_SIZE=1024
```

你也会看到如下的属性：

```
wrapper.java.additional.1=-Delasticsearch-service
wrapper.java.additional.2=-Des.path.home=%ES_HOME%
wrapper.java.additional.3=-Xss256k
wrapper.java.additional.4=-XX:+UseParNewGC
wrapper.java.additional.5=-XX:+UseConcMarkSweepGC
wrapper.java.additional.6=-XX:CMSInitiatingOccupancyFraction=75
wrapper.java.additional.7=-XX:+UseCMSInitiatingOccupancyOnly
wrapper.java.additional.8=-XX:+HeapDumpOnOutOfMemoryError
wrapper.java.additional.9=-Djava.awt.headless=true
```

第 1 个属性负责设置 Elasticsearch 的堆内存的大小，其余的是 JVM 的附加参数。如果想添加另一个参数，则只需增加一个 wrapper.java.additional 属性，其后跟着一个点和下一个可用的数字，例如：

```
wrapper.java.additional.10=-server
```



注意

有一件事必须牢记，垃圾回收器调优不是一件一次性的工作。它需要不断试验，同时它也非常依赖于你的数据、查询和它们的组合。出了问题时不要害怕调整，只要观察它们，看看调整后 Elasticsearch 工作得怎样。

8.2.3 在类 UNIX 系统上避免内存交换

尽管这个不是与垃圾回收和堆内存使用严格相关，但我们认为，了解如何禁止内存交换是重要的。内存交换是把内存页写入磁盘的过程（对于基于 UNIX 的系统指的是交换分

区)。当物理内存的数量不够了或者操作系统由于某些原因认为把一部分 RAM 内存写入磁盘更好些时,就会发生内存交换。如果交换了的内存页再次被需要,操作系统会从交换分区中加载它们,并允许进程使用它们。正如你所想象的那样,这个过程会消耗时间和资源。

当使用 Elasticsearch 时,我们希望避免它的进程内存被交换。你可以想象,如果有部分 Elasticsearch 使用的内存被写到磁盘然后再从磁盘上读取,这会影响搜索和索引的性能。鉴于此, Elasticsearch 允许我们关闭它的内存交换。为了做到这个,你需要在 `Elasticsearch.yml` 文件中设置 `bootstrap.mlockall` 属性为 `true`。

然而,上述设置仅仅是个开始。你还需要通过设置 `Xmx` 和 `Xms` 属性为相同的值来确保 JVM 不会改变堆的大小。可以通过指定 Elasticsearch 的环境变量 `ES_MIN_MEM` 和 `ES_MAX_MEM` 为相同的值来做到这点。还要记得,你需要有足够的物理内存来与你的设置匹配。

现在,如果我们启动 Elasticsearch,我们能够在日志中看到如下的信息:

```
[2013-06-11 19:19:00,858] [WARN ] [common.jna  
Unknown mlockall error 0
```

这意味着我们的内存锁定没有生效。那么现在让我们来修改两个 Linux 操作系统的文件(这需要管理员权限)。假设运行 Elasticsearch 的用户是 `elasticsearch`。

首先,修改 `/etc/security/limits.conf`,添加如下内容:

```
elasticsearch - nofile 64000  
elasticsearch - memlock unlimited
```

其次,修改 `/etc/pam.d/common-session` 文件,添加如下内容:

```
session required pam_limits.so
```

重新使用 Elasticsearch 账户登录后,再次启动 Elasticsearch,你就应该不会看到 `mlockall` 错误了。

8.3 对查询做基准测试

在处理搜索或者数据分析时有一些事情很重要。我们需要结果是精确的,且需要它们是相关的,同时还需要它们尽量快地返回。如果你负责设计在 Elasticsearch 上执行的查询,迟早会发现你处在一个需要优化查询性能的境地。原因有很多,从基于硬件的问题到不良的数据结构,再到糟糕的查询设计。在写作本书时,基准测试 APIHIA 只存在于 Elasticsearch 的分支上,这说明它不是官方 Elasticsearch 分发包中的一部分。目前来说我们要么使用类似 `jMeter` 或者 `ab` (Apache 基准测试工具,参见 <http://httpd.apache.org/docs/2.2/programs/ab.html>),要么使用 Elasticsearch 分支的版本。还要记得现在讨论的功能在最终版本发布时可能会有所改动,所以如果想要使用基准测试功能的话,留意网页 <http://www.Elasticsearch.org/guide/en/Elasticsearch/reference/master/search-benchmark.html> 是个好主意。

8.3.1 为基准测试配置集群

基准测试功能默认是关闭的。任何在正确配置的 Elasticsearch 节点上使用基准测试的尝试都会导致一个类似下面的错误：

```
{
  "error" : "BenchmarkNodeMissingException[No available nodes for
    executing benchmark [benchmark_name]]",
  "status" : 503
}
```

这是正常的，没人希望冒险在生产环境的集群上执行有潜在危险的功能。在进行性能测试和基准测试期间，你会执行许多复杂的和重度的查询，因此在真实用户使用的 Elasticsearch 集群上执行这类的基准测试似乎不是一个好主意。它会导致集群响应缓慢，可以造成系统崩溃和不好的用户体验。为了使用基准测试，你必须告诉 Elasticsearch 哪个节点能够运行需要测试的查询。每个你希望使用基准测试的实例都应该在启动时设置 `--node.bench` 选项为 `true`。例如，可以这样来启动一个 Elasticsearch 实例：

```
bin/elasticsearch --node.bench true
```

另一种方式是添加 `node.bench` 属性到 `Elasticsearch.yml` 文件中，当然，要配置为 `true`。无论选择哪种方式，我们现在可以运行第一个基准测试了。

8.3.2 进行基准测试

Elasticsearch 提供了名为 `_bench` 的 REST 端点，允许我们定义在集群中允许基准测试的节点上执行的任务。让我们通过一个简单的例子来学习怎么做。我们会向你展示一些实际的东西，例如第 2.4 节中讨论过滤器时，我们试图使你相信，在大多数情况下，后向过滤器是不好的。我们现在能够自己检验，看看带有后向过滤器的查询是否真的慢。能够验证这点的命令如下（我们在 Wikipedia 数据库中使用过）：

```
curl -XPUT 'localhost:9200/_bench/?pretty' -d '{
  "name": "firstTest",
  "competitors": [ {
    "name": "post_filter",
    "requests": [ {
      "post_filter": {
        "term": {
          "link": "Toyota Corolla"
        }
      }
    }
  ]
},
{
```

```

    "name": "filtered",
    "requests": [ {
      "query": {
        "filtered": {
          "query": {
            "match_all": {}
          },
          "filter": {
            "term": {
              "link": "Toyota Corolla"
            }
          }
        }
      }
    }
  ]
}

```

向 `_bench` 端点发送的请求的结构非常简单。它包含了一个竞争者列表，Elasticsearch 的基准测试功能会对它们相互比较。一个竞争者是一个查询或者一组查询（因为每个竞争者都能有不止一个查询）。为了便于对结果进行分析，每个竞争者都有它自己的名字。现在来看前面的请求所返回的结果：

```

{
  "status": "COMPLETE",
  "errors": [],
  "competitors": {
    "filtered": {
      "summary": {
        "nodes": [
          "Free Spirit"
        ],
        "total_iterations": 5,
        "completed_iterations": 5,
        "total_queries": 5000,
        "concurrency": 5,
        "multiplier": 1000,
        "avg_warmup_time": 6,
        "statistics": {
          "min": 1,
          "max": 5,
          "mean": 1.95900000000000019,
          "qps": 510.4645227156713,
          "std_dev": 0.6143244085137575,
          "millis_per_hit": 0.0009694501018329939,
          "percentile_10": 1,
          "percentile_25": 2,

```


}

我们的例子相对简单（事实上是非常简单），但是它向你展示了基准测试的有效性。当然了，我们初始的查询没有使用 Elasticsearch 基准测试 API 暴露出来的全部配置项。为了总结所有的选项，我们准备了一个 `bench` 端点的全局可用的选项列表。

- ❑ **name**: 基准测试的名称，它让我们很容易区分多个不同的基准测试（参考下一节）。
- ❑ **competitors**: Elasticsearch 将要执行的测试的定义。它是由描述测试的对象所组成的数组。

- ❑ `num_executor_nodes`: 在测试期间作为查询源的最大 Elasticsearch 节点数。默认为 1 个。
- ❑ `percentiles`: 定义 Elasticsearch 应该计算并在结果中返回的关于查询执行时间的百分数。默认值是 [10, 25, 50, 75, 90, 99]。
- ❑ `iteration`: Elasticsearch 应当为每个竞争者重复执行的次数，默认为 5 次。
- ❑ `concurrency`: 每次迭代的并发数，默认是 5，这意味着 Elasticsearch 将会使用 5 个并发线程。
- ❑ `multiplier`: 在一次迭代中每个查询重复执行的次数，默认重复执行 1000 次。
- ❑ `warmup`: 设置 Elasticsearch 应该进行查询预热。默认执行预热，这意味着这个值被设置为 `true`。
- ❑ `clear_caches`: 默认为 `false`，意味着在每次迭代前，Elasticsearch 不会清理缓存。我们通过设置为 `true` 来改变这点。这个参数同一系列参数相关，这些参数决定哪些缓存需要或者不需要清理。这些额外的参数是 `clear_caches.filter` (过滤器缓存)、`clear_caches.field_data` (字段数据缓存)、`clear_caches.id` (ID 缓存) 和 `clear_caches.recycler` (回收器缓存)。除此之外，还有两个参数接收名称数组，其中 `clear_caches.fields` 指定字段的名称和哪个缓存应当被清理，而 `clear_caches.filter_keys` 指定需要清理的 `filter_key` 的名称。想要获得更多关于缓存的信息，可以参考 6.6 节的内容。

除了全局选项，每个竞争者对象还可以包含如下参数。

- ❑ `name`: 同根级别的名称类似，用来区分不同的竞争者。
- ❑ `requests`: 这项是每个竞争者需要运行的查询的列表。每个对象都是一个使用查询 DSL 定义的标准的 Elasticsearch 查询。
- ❑ `num_slowest`: 需要跟踪的慢查询的数量，默认是 1。如果我们希望 Elasticsearch 跟踪并记录不只一个慢查询，我们可以调高这个参数的值。
- ❑ `search_type`: 指定查询的类型，有限的选项是 `query_then_fetch`、`dfs_query_then_fetch` 和 `count`。默认值是 `query_then_fetch`。
- ❑ `indices`: 索引名称数组，用来限制查询只能在数组中的索引上执行。
- ❑ `types`: 索引类型数组，用来限制查询只能在数组中的类型上执行。
- ❑ `iteration, concurrency, multiplier, warmup, clear_caches`: 这些参数用来覆盖其同名的全局配置。

8.3.3 控制运行中的基准测试

依赖于我们执行基准测试时使用的参数的不同，一个包含一些需要重复几千次的查询的基准测试命令可以运行几分钟或者几个小时。要是能够检查测试运行得如何并预测需要多久测试才能结束就好了。正如你说期望的那样，Elasticsearch 提供了这些信息。为了得到

这些信息，你需要做的仅仅是运行如下的命令：

```
curl -XGET 'localhost:9200/_bench?pretty'
```

上述命令产生的输出可能看起来是这样的（这是在我们用作例子的基准测试执行期间获取的）：

```
{
  "active_benchmarks" : {
    "firstTest" : {
      "status" : "RUNNING",
      "errors" : [ ],
      "competitors" : {
        "post_filter" : {
          "summary" : {
            "nodes" : [
              "James Proudstar" ],
            "total_iterations" : 5,
            "completed_iterations" : 3,
            "total_queries" : 3000,
            "concurrency" : 5,
            "multiplier" : 1000,
            "avg_warmup_time" : 137.0,
            "statistics" : {
              "min" : 39,
              "max" : 146,
              "mean" : 78.95077720207264,
              "qps" : 32.81378178835111,
              "std_dev" : 17.42543552392229,
              "millis_per_hit" : 0.031591310251188054,
              "percentile_10" : 59.0,
              "percentile_25" : 66.86363636363637,
              "percentile_50" : 77.0,
              "percentile_75" : 89.22727272727272,
              "percentile_90" : 102.0,
              "percentile_99" : 124.860000000000013
            }
          }
        }
      }
    }
  }
}
```

感谢这个功能，你能看到测试的进度，并尝试预估在测试完成并返回结果前需要等待多长时间。如果想要放弃当前运行的基准测试（例如，它需要花费太长的时间，并且已经看出测试查询并不是优化的），Elasticsearch 有一个解决方案。例如，为了放弃名称为 `firstTest` 的基准测试，我们在 `_bench/abort` 端口执行一个 POST 请求，如下所示：

```
curl -XPOST 'localhost:9200/_bench/abort/firstTest?pretty'
```

Elasticsearch 的响应会向你展示测试的部分结果。它几乎与前面例子的内容一样，除了基准测试的状态会被设置为 ABORTED 以外。

8.4 热点线程

当你遇到了麻烦，例如你的集群比平时执行得缓慢，使用了大量的 CPU 资源时，那么你需要做些什么来使集群恢复正常。这就是使用热点线程 API 的场景。热点线程 API 能向你提供查找问题根源所必需的信息。这里的一个热点线程是一个 Java 线程，它使用大量 CPU 并且执行了相当长的一段时间。有了这样的一个线程并不意味着 Elasticsearch 本身出了什么问题，它给出了什么可能是热点的信息，并使得你可以看出系统的哪个部分需要更深入的分析，例如查询的执行或者 Lucene 段的合并。热点线程 API 返回从 CPU 的角度来看，Elasticsearch 哪个部分的代码可能是热点的信息，或者由于某些原因 Elasticsearch 卡在了哪里。

当使用热点线程 API 时，通过使用 `/_nodes/hot_threads` 或者 `/_nodes/{node or nodes}/hot_threads` 端点，你可以检查所有的节点、其中的一部分，或者其中一个。例如，为了查看所有节点上的热点线程，我们会执行如下的命令：

```
curl 'localhost:9200/_nodes/hot_threads'
```

这个 API 支持如下参数。

- ❑ **threads**：需要分析的线程数，默认为 3 个。Elasticsearch 通过查看由 `type` 参数决定的信息来选取指定数量的热点线程。
- ❑ **interval**：为了计算线程在某项操作（由 `type` 参数指定）上花费的时间的百分比，Elasticsearch 会对线程做两次检查。我们可以使用 `interval` 参数来定义两次检查的间隔时间。默认为 500ms。
- ❑ **type**：需要检查的线程状态的类型，默认是 `cpu`。这个 API 可以检查线程消耗的时间，线程处于阻塞状态的时间，或者线程处于等待状态的时间。如果你想了解更多关于线程状态的信息，请参考 <http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html>。
- ❑ **snapshots**：需要生成的堆栈跟踪（某一时刻方法调用的嵌入式序列）快照的数量。

使用热点线程 API 十分简单，例如，想要以 1s 为周期查看所有节点上处于等待状态的热点线程，我们会执行如下的命令：

```
curl 'localhost:9200/_nodes/hot_threads?type=wait&interval=1s'
```

8.4.1 热点线程的使用说明

不同于 Elasticsearch 其他的 API，你可以期待返回一个 JSON 数据，热点线程 API 返回

格式化的、包含一系列段落的文本。在我们讨论响应的结构之前，我们想要告诉你一些关于如何产生这个响应的逻辑。Elasticsearch 选取所有运行的线程，并收集关于在每个线程上花费的 CPU 时间的各种信息，例如线程被阻塞或者处于等待状态的次数，处于阻塞或者等待状态持续了多长时间等。然后它会等待一段时间（由 `interval` 参数指定），之后再次收集同样的信息。当这些完成后，对线程基于其消耗的时间进行排序。排序以降序方式进行，这样消耗了最多时间的线程就排在列表的顶部。当然，时间是通过由 `type` 参数指定的操作类型来衡量的。在这之后，前 `N` 个线程（`N` 是由 `threads` 参数指定的线程数）被 Elasticsearch 用来分析。Elasticsearch 做的工作是：每隔几毫秒，对上一步选择的线程获取一些堆栈的快照（快照的数量由 `snapshot` 参数指定）。最后需要做的事情是为了可视化线程状态的变化而组合堆栈信息，然后返回响应给调用者。

8.4.2 热点线程 API 的响应

现在，让我们深入了解一下热点线程 API 的响应。例如，下面的快照是热点线程 API 为刚启动的 Elasticsearch 生成的响应的片段：

```
> curl 'localhost:9200/_nodes/hot_threads'
::: [N'Gabthoth][aBb5552UqvyFck1PNCaJnA][Banshee-3.local][inet[/10.0.1.3:9300]]

1.4% (6.7ms out of 500ms) cpu usage by thread 'elasticsearch[N'Gabthoth][http_server_boss][T#1]{New I/O server boss #51}'
10/10 snapshots sharing following 14 elements
sun.nio.ch.KQueueArrayWrapper.kevent0(Native Method)
sun.nio.ch.KQueueArrayWrapper.poll(KQueueArrayWrapper.java:200)
sun.nio.ch.KQueueSelectorImpl.doSelect(KQueueSelectorImpl.java:103)
sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:87)
sun.nio.ch.SelectorImpl.select(SelectorImpl.java:98)
sun.nio.ch.SelectorImpl.select(SelectorImpl.java:102)
org.elasticsearch.common.netty.channel.socket.nio.NioServerBoss.select(NioServerBoss.java:163)
org.elasticsearch.common.netty.channel.socket.nio.AbstractNioSelector.run(AbstractNioSelector.java:212)
org.elasticsearch.common.netty.channel.socket.nio.NioServerBoss.run(NioServerBoss.java:42)
org.elasticsearch.common.netty.util.ThreadRenamingRunnable.run(ThreadRenamingRunnable.java:108)
org.elasticsearch.common.netty.util.internal.DeadLockProofWorker$1.run(DeadLockProofWorker.java:42)
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
java.lang.Thread.run(Thread.java:744)

0.7% (3.3ms out of 500ms) cpu usage by thread 'elasticsearch[N'Gabthoth][search][T#6]'
10/10 snapshots sharing following 10 elements
sun.misc.Unsafe.park(Native Method)
java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)
java.util.concurrent.LinkedTransferQueue.awaitMatch(LinkedTransferQueue.java:735)
java.util.concurrent.LinkedTransferQueue.xfer(LinkedTransferQueue.java:644)
java.util.concurrent.LinkedTransferQueue.take(LinkedTransferQueue.java:1137)
org.elasticsearch.common.util.concurrent.SizeBlockingQueue.take(SizeBlockingQueue.java:162)
java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1068)
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1130)
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
java.lang.Thread.run(Thread.java:744)

0.5% (2.7ms out of 500ms) cpu usage by thread 'elasticsearch[N'Gabthoth][search][T#10]'
10/10 snapshots sharing following 10 elements
sun.misc.Unsafe.park(Native Method)
java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)
java.util.concurrent.LinkedTransferQueue.awaitMatch(LinkedTransferQueue.java:735)
java.util.concurrent.LinkedTransferQueue.xfer(LinkedTransferQueue.java:644)
java.util.concurrent.LinkedTransferQueue.take(LinkedTransferQueue.java:1137)
org.elasticsearch.common.util.concurrent.SizeBlockingQueue.take(SizeBlockingQueue.java:162)
java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1068)
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1130)
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
java.lang.Thread.run(Thread.java:744)

> |
```

现在我们来解释响应的各个部分。为此我们会使用一个与前面展示的响应有轻微区别的响应。我们这么做是为了更好地可视化在 Elasticsearch 中发生了什么，请记住响应的总体结构不会改变。

热点线程 API 响应的第 1 部分向我们展示线程在哪个节点上。例如，第 1 行的响应可能是下面这样：

```
::: [N'Gabthoth] [aBb5552UQvyFck1PNCaJnA] [Banshee-3.local] [inet[/10.0.1.3:9300]]
```

我们能看到热点线程 API 返回的信息是关于哪个节点的，这在向多个节点发送热点线程请求时非常有用。

热点线程 API 响应接下来的内容可以分成多个小节，每个小节由类似下面的行开始：

```
0.5% (2.7ms out of 500ms) cpu usage by thread  
'elasticsearch[N'Gabthoth] [search] [T#10]'
```

在我们的例子中，我们看到一个名为 search 的线程，在统计结束时，它消耗了百分之 0.5 的 CPU 时间。其中的 cpu usage 部分表明我们在使用 type 等于 cpu 的方式进行统计。在这里你还可能会看到表示线程处于阻塞状态的 block usage 和表示线程处于等待状态的 waiting usage。这里线程的名称十分重要，因为通过查看线程名称，我们可以知道 Elasticsearch 的哪个功能是热点。在我们的例子中，我们知道这个线程是关于搜索的（名称中的 “search”）。你可以期望看到的其他值是 recovery_stream（回复模块事件）、cache（缓存事件）、merge（索引段合并线程）、index（数据索引线程）等。

热点线程 API 响应的下一个部分是以下面内容开头的小节：

```
10/10 snapshots sharing following 10 elements
```

这个信息后面会跟着一个堆栈跟踪信息。在我们的例子里，10/10 表示对同一个堆栈跟踪生成了 10 个快照。通常，这意味着全部的检查时间被花费在了 Elasticsearch 节点的同一个部分。

8.5 扩展Elasticsearch

正如我们在本书和《Elasticsearch Server, Second Edition》中多次说过的，Elasticsearch 是一个高度可扩展的搜索和分析平台。我们可以在水平和垂直方向上对 Elasticsearch 进行扩展。

8.5.1 垂直扩展

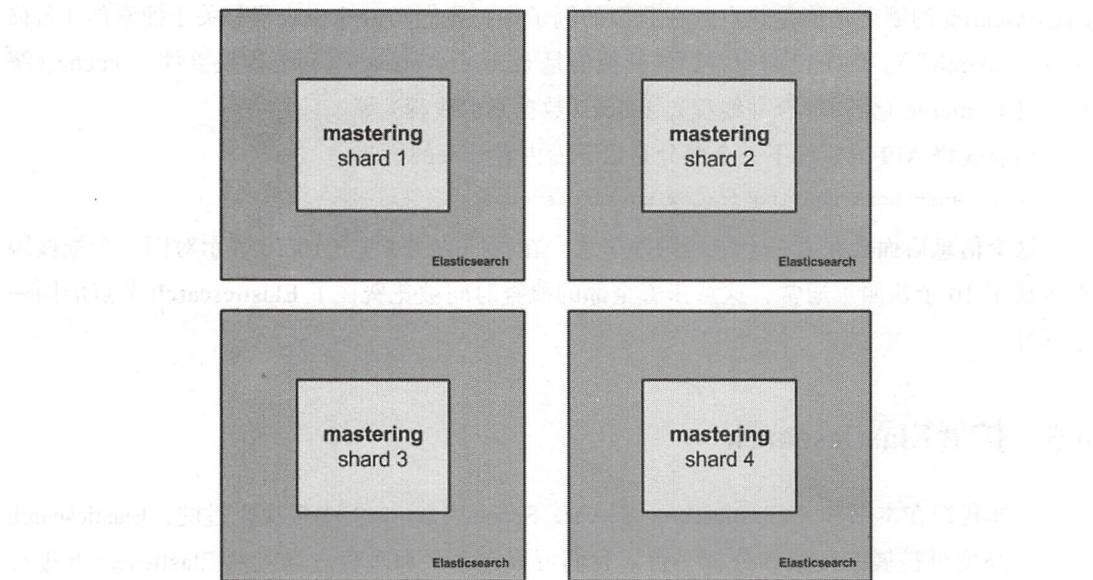
当我们说到垂直扩展时，通常意味着向运行 Elasticsearch 的服务器添加更多的资源：可以添加内存，可以更换到有着更佳的 CPU 或者更快的磁盘存储的机器上。显然，使用更

好的机器，我们可以期望性能的提升。依赖于我们的部署环境和它的瓶颈，可以有较小或则较高的提升。然而，垂直扩展有着它的限制，例如，服务器上的最大可用物理内存或者 JVM 需要使用的总内存。当你有足够多的数据和复杂的查询时，你会很快碰到内存问题，添加新的内存也许帮不了你。

例如，由于垃圾回收和无法使用压缩选项（这意味着为了标记相同的内存空间，JVM 需要使用 2 倍的内存），你不会想要给 JVM 分配超过 31GB 的物理内存。尽管这看起来是一个大问题，垂直扩展并不是唯一的解决方案。

8.5.2 水平扩展

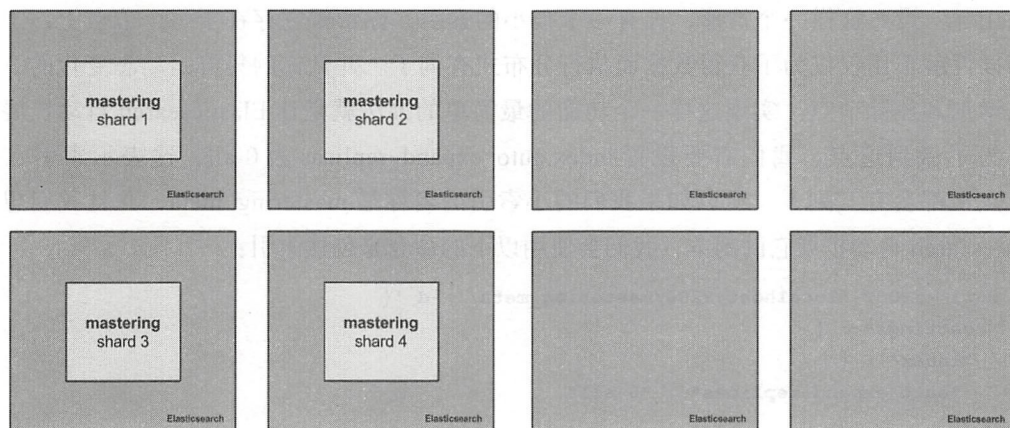
对于 Elasticsearch 用户来说，另一个解决方案是水平扩展。相对来说垂直扩展就像是建造一个摩天大楼，而水平扩展就像在住宅区内建造多所房子。我们选择使用多台机器并将数据分割存储在其上，以此来替代投资硬件和购买更好的机器。水平扩展给了我们几乎无限的扩展能力。即便使用了最好的硬件，用一台机器来容纳数据和处理查询也是不够的。如果一台机器容纳不下数据，我们会把索引分成多个分片（shard），并在集群中分散它们，就像下图所展示的那样：



当没有足够的计算能力来处理查询时，总是可以为分片增加更多的副本。上图的集群有 4 个节点，其上运行着由 4 个分片构成的 `mastering` 索引。

如果想要增加集群处理查询的能力，只需增加额外的节点，例如 4 个。增加节点后，既可以创建有着更多分片的新索引来平衡负载，也可以给现有的分片增加副本。两种方案

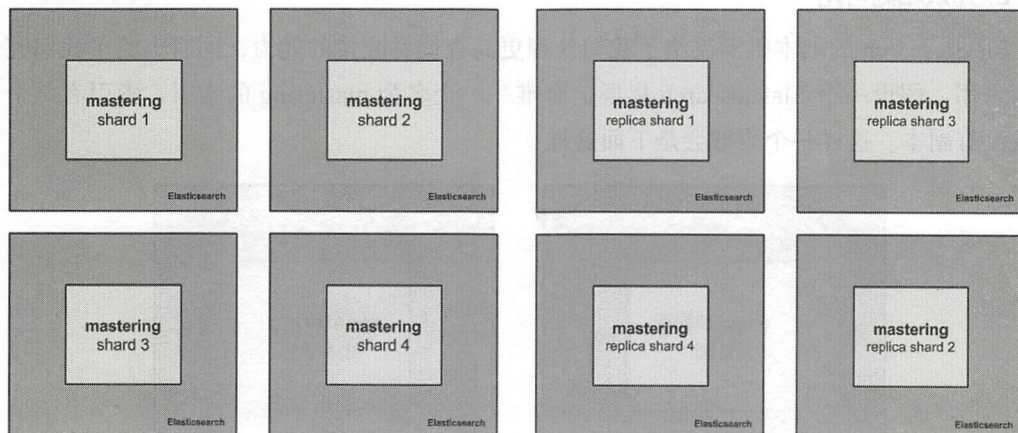
都是可行的。当硬件容纳不下数据时，应该寻求更多的主分片。在这种情形下，我们通常会碰到存储器溢出的情形、分片查询时间变长、内存交换或者大量的 I/O 等待。第 2 个方案在我们的硬件能够很好地处理数据，但是流量过高以致于节点无法跟上时使用。第 1 个方案比较简单，我们来看看第 2 个。有着 4 个额外的节点，我们集群如下图所示：



现在，让我们通过执行下面的命令来添加一个副本：

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{  
  "index" : {  
    "number_of_replicas" : 1  
  }  
}'
```

我们的集群现在差不多是下面的样子：



我们可以看到，每个构成 mastering 索引的初始分片都有一个副本保存在其他节点上。

于是 Elasticsearch 便能够在分片和它们的副本间做负载均衡了，进而查询便不会总是命中一个节点了。于是我们能够处理两倍于初始部署方式的查询负载。

1. 自动创建副本

Elasticsearch 允许我们在集群足够大时自动扩展副本数。你可能会奇怪，这个功能有什么用呢？设想这样一个情形，你有一个很小的索引，你希望它存在于每个节点上，于是你的插件就不用仅仅为了获得数据而执行分布式查询了。并且你的集群是动态变化的，你可以增加和删除节点。实现这样一个功能的最简单的办法就是让 Elasticsearch 自动扩展副本。为了做到这点，我们需要设置 `index.auto_expand_replicas` 为 `0-all`，这表示索引在其他节点上都会有其副本。所以如果我们的小索引的名称是 `mastering_meta`，并且我们想让 Elasticsearch 自动扩展它的副本，我们会使用以下的命令来创建索引：

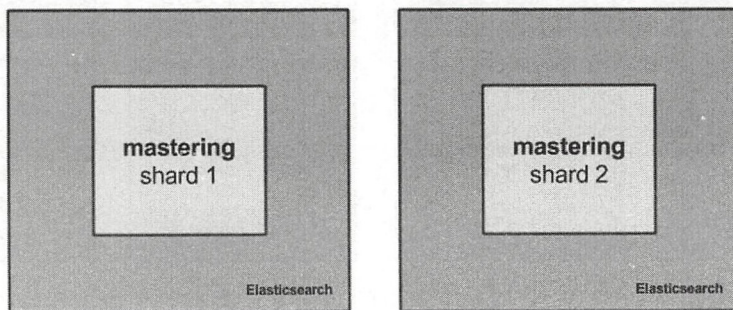
```
curl -XPOST 'localhost:9200/mastering_meta/' -d '{
  "settings" : {
    "index" : {
      "auto_expand_replicas" : "0-all"
    }
  }
}'
```

如果索引已经存在，我们也可以使用下面的命令来更新索引的配置：

```
curl -XPUT 'localhost:9200/mastering_meta/_settings' -d '{
  "index" : {
    "auto_expand_replicas" : "0-all"
  }
}'
```

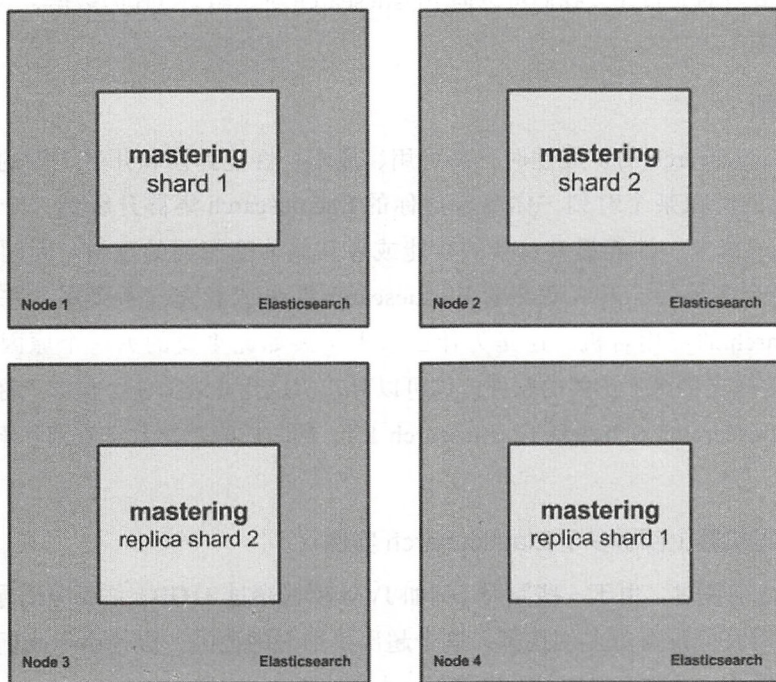
2. 冗余和高可用

Elasticsearch 的副本机不仅给了我们处理更高查询吞吐量的能力，同时也给了我们冗余和高可用。假设一个 Elasticsearch 集群上有唯一一个名为 `mastering` 的索引，索引有两个分片且没有副本。这样一个集群会是下面这样：



现在，当有一个节点宕掉后会发生什么呢？简单讲，我们会丢失 50% 的数据，且如果故障是致命的，我们会永远丢失这些数据。即便有备份，我们也需要引入一个新节点并恢复备份。这需要时间。如果你的业务依赖于 Elasticsearch，故障停机意味着金钱的损失。

现在，让我们看看同样的集群但是有一个副本的情况：



现在失去一个节点意味着我们仍然拥有完整可用的数据，且我们可以不停止服务就能恢复完整的集群结构。并且这样部署的话，我们可以在某些情形下忍受 2 个节点同时失效。例如，节点 1 和节点 3 或者节点 2 和节点 4。在这两种情形下我们仍然可以访问全部数据。当然这会降低性能，因为集群缺失了节点，但这仍然比完全不响应查询要好。

于是，当你在设计架构、决定节点数量、有多少个索引以及每个索引的分片数量时，你需要把能接受的出现故障的节点数量考虑进去。当然了，你还需要考虑性能，只不过冗余和高可用应该是进行扩展时的一个因数。

3. 成本和性能的适应性

Elasticsearch 天然的分布式特征和能够水平扩展的能力让我们在面对运行时的性能和成本问题时很容易适应。首先，有着高性能的磁盘、众多的 CPU 和大量的内存的高端服务器价格昂贵。并且云计算越来越流行，不仅允许我们在租来的机器上部署运行，还允许我们按需进行扩展。我们只需要增加更多的机器，这只需点击几下鼠标或者经过一些配置，甚

至可以被自动化。

综合以上内容，当 Elasticsearch 有了水平扩展的解决方案后，我们可以降低运行集群的成本。并且，如果成本对于我们的商业计划是最重要的因数，我们可以很容易地通过牺牲性能来应对。当然，我们也可以选择其他方式。如果我们能负担起庞大的集群，使用适当的硬件和适当的分布式，我们可以向 Elasticsearch 推送数百 TB 的数据，并且仍然能够获得不错的性能。

4. 持续更新

在讨论 Elasticsearch 的扩展性时，高可用、成本、性能弹性和几乎无限的增长并不是唯一需要讨论的。在某个时刻，你会希望你的 Elasticsearch 集群升级到一个新版本，这可能是由于 bug 修复、性能提升、新的功能或者其他你能想到的理由。问题是当每个分片都只有唯一实例时，升级意味着 Elasticsearch 部分或者完全不可用，并可能意味着使用 Elasticsearch 的应用宕机。这是为什么水平扩展如此重要的另一个原因。对于类似 Elasticsearch 这样支持水平扩展的软件，你可以对它们执行升级操作。例如，你可以通过滚动重启把 Elasticsearch 1.0 升级到 Elasticsearch 1.4，同时所有数据对于查询和索引操作仍然是可用的。

5. 一个物理机器上部署多个 Elasticsearch 实例

尽管我们之前说过，由于一些原因（例如 JVM 堆栈超过 31GB）你不应该寻求最高性能的机器，但有时我们没有更多的选择。这个超出了本书的范围，但是由于我们在讨论扩展性，我们认为提及一下在这种情况下应该怎么办是有益的。

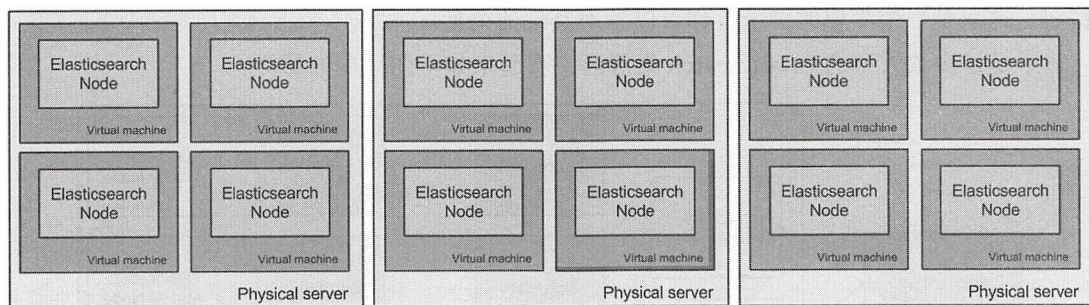
在遇到我们正在讨论的情形时，当我们有高端的硬件，配备大量的内存、许多的高速硬盘、众多的 CPU 等，我们需要考虑把物理服务器分割为多个虚拟机器，然后在每个虚拟机上运行一个 Elasticsearch 实例。



注意

不运行多个虚拟机而是直接在一个物理机上运行多个 Elasticsearch 实例也是可行的。选择哪个方案取决于你。不过，我们喜欢把事情分开，由此，我们通常会把服务器分割成多个小虚拟机。在把服务器分割为多个小虚拟机时需要记住，这些小虚拟机会共享 I/O 子系统。于是，为虚拟机恰当地分配磁盘是一个好的选择。

为阐明这样的部署，请看下图。它显示了你如何在 3 台大型服务器上以云的方式运行 Elasticsearch，每台服务器都被分成了 4 个独立的虚拟机。每个虚拟机负责运行一个 Elasticsearch 实例。



6. 阻止分片及其副本部署在同一个节点上

还有一件事情需要讲一下。当有多个物理服务器被分割成虚拟机时，确保分片和它的副本不在同一台物理机上十分重要。那样的话如果一个服务器出现故障或者重新启动就会发生悲剧了。我们可以使用集群部署感知告诉 Elasticsearch 分离分片和副本。对于前面的例子，我们有 3 台物理服务器，分别命名为 server1、server2 和 server3。

现在对于一个物理服务器上的每个 Elasticsearch，我们定义 `node.server_name` 属性并设置为服务器的标识符。于是对于在第 1 台物理服务器上的所有 Elasticsearch 节点来说，我们会在 `Elasticsearch.yml` 文件中设置如下属性：

```
node.server_name: server1
```

除此之外，每个 Elasticsearch 节点（无论在哪个物理服务器上）需要向 `Elasticsearch.yml` 文件中添加如下属性：

```
cluster.routing.allocation.awareness.attributes: server_name
```

这告诉 Elasticsearch 不要把主分片和它的副本放到具有相同 `node.server_name` 属性的节点上。我们只需完成这个就足够了，Elasticsearch 会完成后续的工作。

7. 为大规模集群设计节点的角色

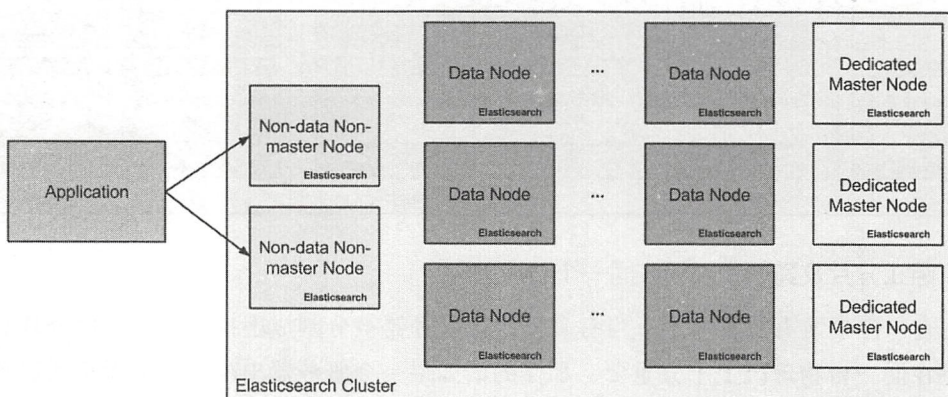
我们还想告诉你一件事；事实上，我们已经在本书和《Elasticsearch Server, Second Edition》中提过了，为了有一个完全容错和高可用的集群，我们应该区分节点，并给每个节点一个设计好的角色。我们可以给每个 Elasticsearch 节点指派的角色如下：

- ☐ 查询聚合节点
- ☐ 数据节点
- ☐ 候选主节点

默认时，每个 Elasticsearch 节点都是候选主节点（它可以成为主节点），能够容留数据，以及作为查询聚合节点。查询聚合节点可以发送粒子查询到其他节点，收集和合并结果，以及响应发出查询的客户端。你可能会奇怪为什么需要这个。让我们来举个简单的例子：如果主节点有很大的压力，它可能不能及时处理集群状态相关的命令，于是集群将变得不

稳定。这只是一个简单的例子，你可以考虑其他的情形。

于是，多数大型的 Elasticsearch 集群通常看起来如下图呈现的那样：



正如你所见到的，这个假设的集群包含两个聚合节点（因为我们知道不会有太多的查询，但是我们希望有冗余），数十个数据节点（因为数据量非常大），以及至少 3 个候选主节点（不应该做其他的事情）。Elasticsearch 在任意给定的时刻只需要使用一个主节点，为什么这里有 3 个呢？这是用于冗余，同时通过设置 `discovery.zen.minimum_master_nodes` 为 2 也能阻止脑分裂的发生。这使得我们很容易地处理集群中某个候选主节点出现故障的情形。

现在，让我们给你一些集群中每个节点类型的配置片段。虽然我们已经在 7.1 节中讨论过了，但我们想再讲一次。

（1）查询聚合节点

查询聚合节点的配置十分简单。为了配置它们，我们只需要高 Elasticsearch 我们不希望这些节点成为候选主节点以及容纳数据。对应的 `Elasticsearch.yml` 中的配置如下：

```
node.master: false
node.data: false
```

（2）数据节点

数据节点配置起来同样简单：我们只需要声明它们不应该是候选主节点即可。然而，我们不是默认配置的拥趸（因为默认配置趋于变化），因此数据节点的配置如下：

```
node.master: false
node.data: true
```

（3）候选主节点

我们把候选主节点留到了通用扩展小节的最后。当然，这类的 Elasticsearch 节点不应该容纳数据，但是除此之外，在这类节点上禁用 HTTP 协议也是一个好的实践。这样做是为了避免意外地在这些节点上执行查询。候选主节点相比数据节点和查询聚合节点可以使用更少的资源，于是我们需要确保它们仅仅被用来处理与主节点相关的工作。所以候选主节点的配置看起来大概是下面的样子：

```
node.master: true
node.data: false
http.enabled: false
```

8.5.3 在高负载的场景下使用 Elasticsearch

现在我们已经知道了理论（以及一些 Elasticsearch 扩展的例子），我们已经准备好讨论为应对高负载需要对 Elasticsearch 进行哪些方面的准备。我们决定把本章的这个部分分成 3 个小节：一个专注于高索引负载，一个专注于高查询负载，一个同时考虑这两种情况。这会给你一些在你准备你的集群时需要考虑什么的建议。

考虑在配置好为生产环境使用的集群后进行性能测试。不要直接使用从本书中得到的指标。使用你的数据和你的查询进行尝试和调整，并观察它们的区别。记住给出对每个人都适用的建议是不可能的，所以，应该把下面两部分当作一般的建议而不是可以使用的准则。

1. Elasticsearch 优化的常规建议

在本节中，我们会讨论 Elasticsearch 调优相关的常规建议。它们不是只与索引性能或者查询性能相关，而是与它们都相关。

（1）选择正确的存储

关键点之一是我们选择正确的存储实现。这在运行 Elasticsearch 1.3.0 之后的版本时尤其重要。通常，如果你使用一个 64 位的操作系统，你应该使用 mmapfs。如果你没有使用 64 位操作系统，那么，基于 UNIX 系的系统应该选择 niofs，基于 Windows 的系统应该选择 simplefs。如果你能接受一个快速但是非持久化的存储，你可以看看 memory 存储。它会带给你最佳的索引访问性能，但是需要足够的内存来处理不仅全部的索引文件，还要处理索引和查询请求。

随着 Elasticsearch 1.3.0 的发布，我们有了一个新的名为 default 的存储类型，它是最新的默认存储类型。正如 Elasticsearch 开发者所说，它是一个混合的存储类型。它使用内存映射文件来读取 term 字典和 doc values，而其他的文件则使用 NIOFSDIRECTORY 实现来访问。在大多数情况下，当使用 Elasticsearch 1.3.0 或者更高的版本时，应该使用 default 存储类型。

（2）索引刷新频率

我们需要留意的第 2 件事情是索引刷新频率。我们知道索引刷新频率是指文档需要多长时间才能出现在搜索结果中。规则非常简单：刷新频率越短，查询越慢，且索引文档的吞吐量越低。如果我们能够接受一个较慢的刷新频率，例如 10 秒或者 30s，那设置成这样是十分有益的。这会减轻 Elasticsearch 的压力，因为内部对象会以一个较慢的速度重新打开，于是会有更多可用的资源来处理索引和查询请求。请记住，刷新频率默认是 1s，这基本上意味着索引查询器每 1s 重新打开一次。

为了让你了解对于我们正在讨论的问题有什么性能上的收获，我们做了一些性能

测试，观察 Elasticsearch 在不同的刷新频率下的性能。当刷新频率为 1s 时，使用一个 Elasticsearch 节点我们大概每秒能够索引 1000 个文档。当提高刷新频率为 5s 时，索引吞吐量提升了 25% 以上，我们大概每秒可以索引 1280 个文档。当设置刷新频率为 25s 时，相对刷新频率 1 秒的情形吞吐量提升了超过 70%，大约每秒索引 1700 个文档。同样值得一提的是，无限增加刷新时间是没有意义的，因为超过一定值（取决于你的数据负载和数据量）之后，性能提升变得微乎其微。

（3）线程池调优

这是与你的部署环境紧密相关的事情。一般而言，Elasticsearch 默认的线程池配置就足够优化了。然而，总有一些时候这些配置不能满足实际需要。你需要牢记，仅在遇到以下的情形时才调整默认的线程池配置，即你看到节点正在填充队列并且仍然有计算能力剩余，且这些计算能力可以被指定用于处理待处理操作。

例如，如果你在做性能测试时发现 Elasticsearch 实例没有 100% 饱和，但是你却接到了拒绝执行错误，那么这时就需要调整 Elasticsearch 线程池了。你既可以增加同时执行的线程数，也可以增加队列的长度。当然，你需要知道增加并发执行的线程数到一个很大的数值时，会产生大量的 CPU 上下文切换（http://en.wikipedia.org/wiki/Context_switch），进而导致性能下降。当然，大量的队列也不是一个好主意，通常快速报错要比用队列中的成千上万的请求来压垮 Elasticsearch 要好。然而，这些都取决于你的特定的部署环境和使用场景。我们很想给你一个精确的数字，但是对于这个问题，是无法给出的。

（4）调整合并过程

Lucene 段合并的调整是另一件高度取决于你的使用场景和一些相关因素的事情。这些因素，如你追加多少数据，多久追加一次等。对于 Lucene 分段和合并需要记住两件事情。在有多段索引上执行查询要比在只有少量段的索引上执行慢。性能测试显示在由多个段构成的索引上执行查询比在只有一个段的索引上慢大约 10% 到 15%。另一方面，尽管段合并不是免费的，我们越希望有更少的段，就越需要配置激进的段合并策略。

一般的，如果你希望查询更快，就寻求更少的索引段。例如，设置 `index.merge.policy.merge_factory` 低于默认值 10，会导致更少的段，更低的 RAM 消耗，更快的查询执行速度和更慢的索引速度。设置 `index.merge.policy.merge_factory` 高于默认值 10，会导致索引由更多的段来构成，更高的 RAM 消耗，更慢的查询速度和更快的索引速度。

还有一件事：限流。默认条件下，Elasticsearch 会限制合并的速度在 20MB/s。Elasticsearch 使用限流来避免合并过程过多的影响搜索。并且，如果合并过程不是足够快的话，Elasticsearch 会限制索引过程只使用一个线程，以此来确保合并过程能够顺利结束，且没有大量的段存在。然而，如果你在使用 SSD 硬盘，那么限流 20MB/s 就不合适了，你可以把它放大 5 到 10 倍。想要调整限流，我们需要在 `Elasticsearch.yml` 文件里（或者使用集群配置

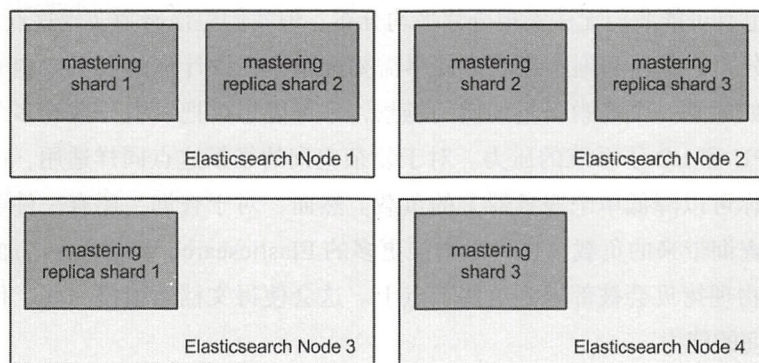
API) 设置 `indices.store.throttle.max_bytes_per_sec` 属性为一个期望值, 例如 200MB/s。

通常而言, 如果你想要更快的索引文档, 就寻求个多的索引段。如果你想要查询更快, 由于合并你的 I/O 系统能完成更多的工作, 且你能接受 Elasticsearch 消耗多一点的内存, 那么就配置激进一些的合并策略。如果你希望 Elasticsearch 索引更多的文档, 就配置不那么激进的合并策略, 但是要记得这会查询的性能。如果你两个都想要, 那么你需要寻找一个平衡点, 在这个点上合并既不会太频繁同时也不会导致大量的段。

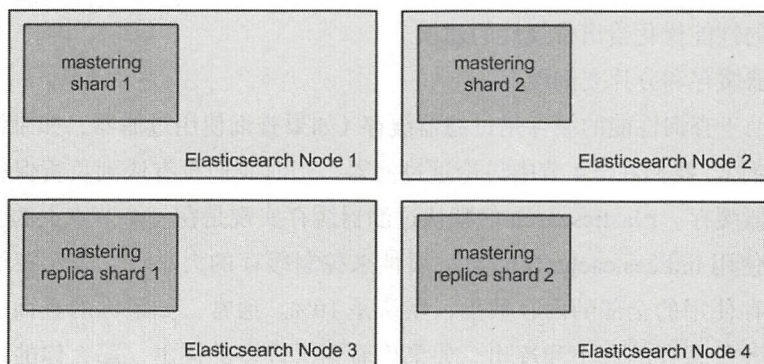
(5) 数据分布

我们已经知道, Elasticsearch 的每个索引都可以被分成多个分片, 并且每个分片都能有多个副本。当你有不只一个 Elasticsearch 节点和分割了分片的索引, 适当的数据分布对于均衡集群的负载以及没有节点做了比其他节点更多的工作就是十分重要的了。

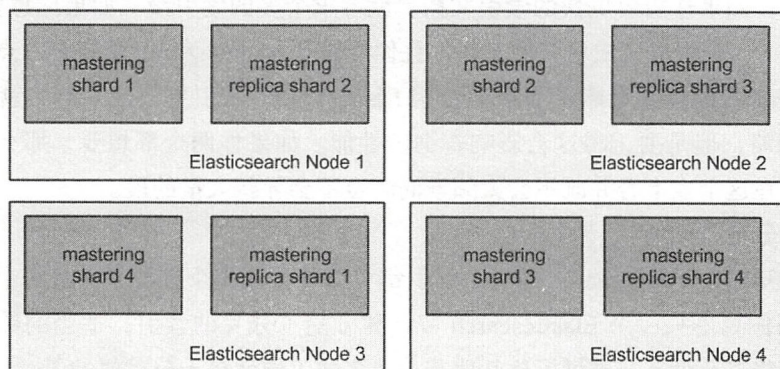
让我们看下接下来的例子。假设我们有一个由 4 个节点构成的集群, 其上有一个由 3 个分片组成和一份副本的索引。集群看起来如下图:



从上图中可以看到, 前两个节点上有两个分片部署在其上, 而后两个只有一个分片。所以实际的部署并不平均。当发送查询和索引数据时, 会使得前两个节点比后两个做更多的工作。这是我们希望避免的。我们可以让 `mastering` 索引有两个分片和一个副本, 于是看起来如下图:



或者，我们可以把 mastering 索引分成 4 个分片和 1 个副本。



对于以上两种方案，我们都得到了平均分布的分片和副本，每个节点都完成相似数量的工作。当然，在有更多的索引（例如按日创建的索引）时，为了使数据平均分布就需要更多的技巧了。也有可能我们无法实现分片平均分布，但我们应该努力实现这点。

对于数据分布、分片和副本还有一件事需要记得，在设计你的索引架构时，你需要记得你的目标。如果是一个高索引量的使用场景，你可能想要把索引分散到多个分片上来降低服务器的 CPU 和 I/O 子系统的压力。对于复杂查询的场景这点同样适用，因为通过使用更多的分片，你可以降低单个服务器上的负载。然而，对于查询，还有一件事：如果你的节点无法处理查询带来的负载，你可以增加更多的 Elasticsearch 节点，并增加副本的数量，于是主分片的物理拷贝会被部署到这些节点上。这会使得文档索引慢一些，但是会给你同时处理更多查询的能力。

2. 高查询频率场景下的建议

Elasticsearch 的一个强大的功能是能够搜索和分析索引过的数据。然而，有时用户需要调整 Elasticsearch，查询不仅要返回结果，还要尽快返回（或者在一个合理的时间范围内）。在这部分，我们不仅讨论可行性，而且也对 Elasticsearch 的高查询吞吐量场景进行优化。我们还会对查询的性能优化给出常规性的建议。

（1）过滤器缓存和分片查询缓存

第 1 个有助于查询性能的缓存是过滤器缓存（如果查询使用过滤器，如果没使用，应该适当地使用它们）。我们在 2.4 节中讨论过过滤器。当时我们没有谈到负责保存过滤器结果的缓存：过滤器缓存。Elasticsearch 的默认过滤器缓存实现是在一个节点上的全部索引间共享，我们可以使用 `indices.cache.filter.size` 属性来控制缓存的大小。它表示在给定节点上能够被过滤器缓存使用的全部的内存数量，默认是 10%。通常，如果你的查询已经使用了过滤器，你应该监控缓存的大小和逐出。如果你看到了许多的逐出，那么你的缓存大概太小

了，你应该考虑增加缓存的大小。缓存过小可能会影响查询的性能。

Elasticsearch 引入的第 2 个缓存是分片查询缓存。它在 Elasticsearch 的 1.4.0 版时引入，它的目的是缓存聚合、提示词结果和命中数（它不会缓存返回的文档，因此，它只在 `search_type=count` 时起作用）。当你的查询使用了聚合或者提示词，最好启用这个缓存（它默认是关闭的），于是 Elasticsearch 就能够重用保存在里面的数据了。关于这个缓存的最好的一点是，它承诺与没有使用缓存时一样准实时搜索。

想要开启分片查询缓存，我们需要设置 `index.cache.query.enable` 属性为 `true`。例如，想要开启 `mastering` 索引的缓存，我们可以使用如下的命令：

```
curl -XPUT 'localhost:9200/mastering/_settings' -d '{
  "index.cache.query.enable": true
}'
```

请记得如果我们不使用聚合或者提示词，那么使用分片数据缓存就毫无意义。

还有一点需要注意，分片查询缓存默认使用不超过分配给 Elasticsearch 节点的堆栈的 1% 的内存。想要改变默认值，我们可以使用 `indices.cache.query.size` 属性。通过使用 `indices.cache.query.expire` 属性，我们可以指定缓存的过期时间，但是这并不是必需的，在大多数情况下，结果保存在缓存中，在每次索引刷新操作后失效。

（2）关于查询的思考

我们能给出的最通用的建议是：你应该总是考虑到优化查询结构、过滤器的使用等。我们在第 2 章中用大量篇幅讨论了这个，但是我们还想要再提及一次，因为我们认为它非常重要。例如，我们看下面的查询：

```
{
  "query" : {
    "bool" : {
      "must" : [
        {
          "query_string" : {
            "query" : "name:mastering AND department:it AND
              category:book"
          }
        },
        {
          "term" : {
            "tag" : "popular"
          }
        },
        {
          "term" : {
            "tag" : "2014"
          }
        }
      ]
    }
  }
}
```

```

    ]
  }
}
}

```

它返回了匹配查询条件的图书的名字。然而，对于前面的查询我们还有一些事情可以优化。例如，我们可以移动一些东西到过滤器，于是下次我们再使用这个查询的某部分时，我们节省了 CPU 并重用了缓存中保存的信息。例如，以下是优化后的查询的样子：

```

{
  "query" : {
    "filtered" : {
      "query" : {
        "match" : {
          "name" : "mastering"
        }
      },
      "filter" : {
        "bool" : {
          "must" : [
            {
              "term" : {
                "department" : "it"
              }
            },
            {
              "term" : {
                "category" : "book"
              }
            },
            {
              "terms" : {
                "tag" : [ "popular", "2014" ]
              }
            }
          ]
        }
      }
    }
  }
}

```

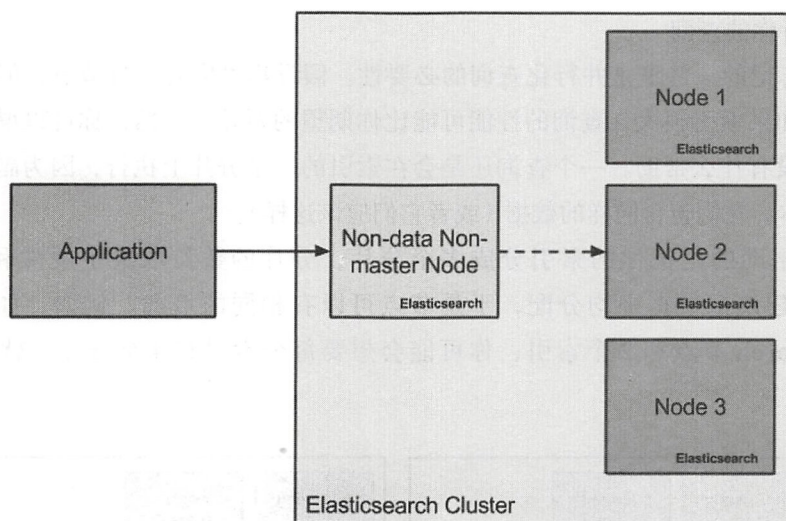
你可以看到，我们做了一些改变。首先，我们使用 `filtered` 查询来引入过滤器，我们移动了绝大多数静态的、不分词的字段到过滤器。这使得我们很容易在执行下一个查询时重用它们。由于进行了这样的查询重构，我们能够简化主查询，所以我们将 `query_string` 查询改为 `match` 查询，因为对于我们的例子来说 `match` 查询就足够了。你在优化查询或者设计它们是需要做的，就是在头脑中思考优化和性能并尽可能地做到最优。

然而，对于查询的产出来说，性能并不是唯一的不同。你知道，过滤器不影响返回文

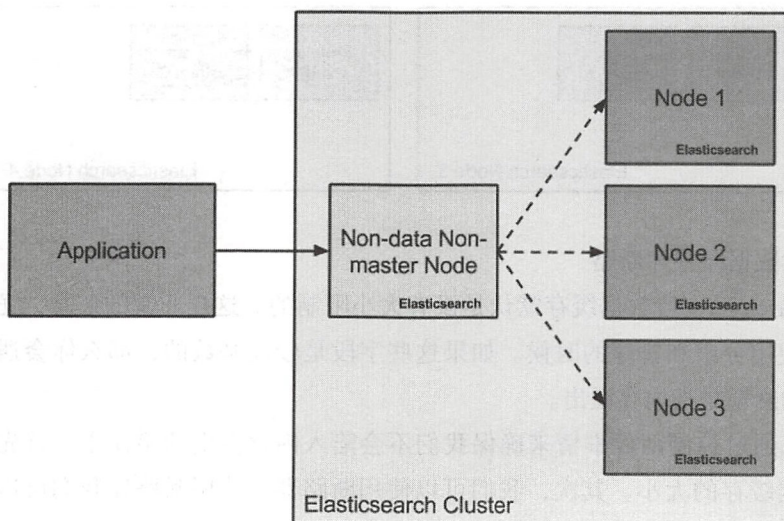
档的打分，并且在计算得分时不被考虑进去。于是，如果你对比前面查询返回的文档的得分，你会注意到它们是不同的。这点需要记住。

（3）使用路由

如果你的数据可以使用路由，你应该考虑使用它。有着相同路由值的数据都会保存到相同的分片上。于是，我们可以避免在请求特定数据时查询所有的分片。例如，如果我们保存客户的数据，我们可以使用客户 ID 作为路由。这会允许我们把同一客户的数据保存在同一个分片上。这意味着在查询时，Elasticsearch 只需从一个分片上获取数据，如同下图所展示的那样：



如果我们假设数据保存在节点 2 上，我们可以看到 Elasticsearch 只需要在特定的节点上执行查询就能获取指定客户的所有数据。如果我们不使用路由，前面查询的执行起来就会如下：



在不使用路由的情形下，Elasticsearch 先是需要搜索全部的分片。如果你的索引包含数十个分片，如果使用路由，只要单个 Elasticsearch 实例还能够容纳得下分片，性能的提升就会非常明显。

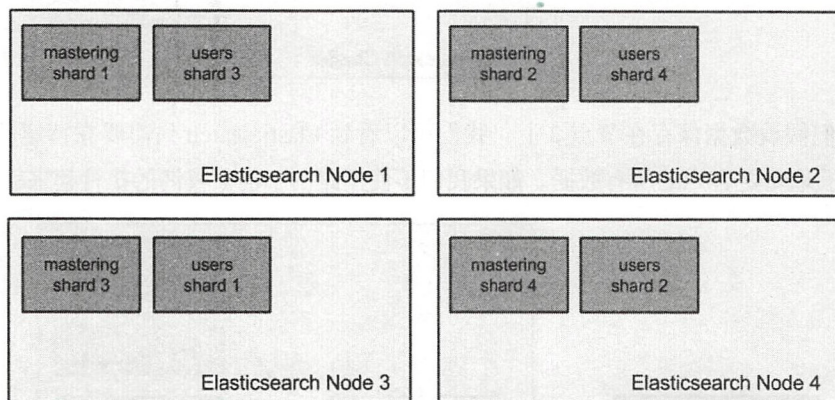


注意 请记得，不是每种情况都适合使用路由。想要使用它，你的数据需要是可分割的，这样它们才能够分布在不同的分片上。例如，拥有数十个非常小的分片和一个庞大的分片是没有意义的，因为对于这个庞大的分片性能不会很好。

(4) 并行你的查询

通常被忘记的一件事是并行化查询的必要性。假设集群中有一打节点，但是索引只有一个分片。如果索引很大，查询的性能可能比你期望的要差。当然，你可以增加副本的数量，但是这没有什么帮助。一个查询还是会在索引的一个分片上执行，因为副本只不过是主分片的副本，它们包含同样的数据（或者它们应该这样）。

真正有帮助的是把你的索引分成多个分片，分片的数量取决于硬件和部署方式。一般来说，建议把数据平均分配，于是节点可以有相同的负载。例如，如果我们有 4 个 Elasticsearch 节点和 2 个索引，你可能会想要每个索引有 4 个分片，就像下图所示的那样：



(5) 字段数据缓存和断路

Elasticsearch 的字段数据缓存默认是没有大小限制的。这样非常的危险，尤其是当你在很多字段上使用分组和排序的时候。如果这些字段是较高基数的，那么你会遇到更大的麻烦。我们说的麻烦是指内存溢出。

我们可以通过控制两件事情来确保我们不会陷入内存溢出的错误中。首先，我们可以限制字段数据缓存的大小。其次，我们可以使用断路器。使用断路器我们可以轻易地配置

成直接抛出异常而不是加载过多的数据。结合这两种办法就能确保我们不会遇到内存溢出的问题。

但是，我们也要记住，Elasticsearch 在字段数据缓存的大小不够处理分组或者排序请求时会逐出数据。这会影响查询的性能，因为加载字段数据信息的效率不是很高。然而，我们认为让查询慢点总比让集群由于内存溢出错误而宕掉要好。

最后，如果你的查询大量使用字段数据缓存（例如聚合和排序），且你遇到了内存相关的问题（例如内存溢出错误或者 GC 停顿），可以考虑使用我们讨论过的 `ocValue`。`doc value` 有着同字段数据缓存相似的性能，并且随着 Elasticsearch 每次新版的发布对其的支持也越来越好（对 `doc value` 的改进是由 Lucene 来实现的）。

（6）控制 `size` 和 `shard_size`

在处理使用聚合的查询时，对于某些查询我们可以使用两个属性：`size` 和 `shard_size`。`size` 参数定义最后的聚合结果会返回多少组数据。聚合最终结果的节点会从每个返回结果的分片获取靠前的结果，且只会返回前 `size` 个结果给客户端。`shard_size` 参数具有相同的含义，知识其作用在分片层次上。增加 `shard_size` 会导致聚合结果更加准确（例如对重点词的聚合），代价是更大的网络开销和内存使用。降低这个参数会导致聚合的结果不那么精确，但却有着网络开销小和内存使用低的好处。如果我们看到内存使用太多了，我们可以降低问题查询的 `size` 和 `shard_size` 参数，看看结果的质量是否仍然可以接受。

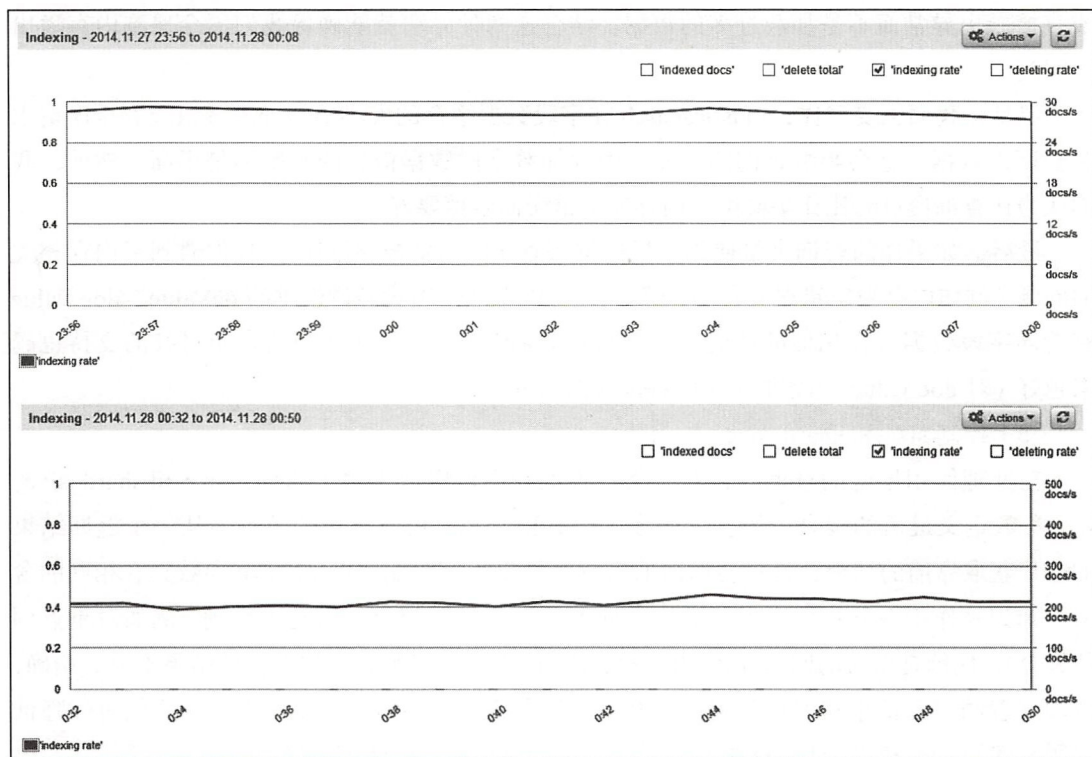
3. 高索引吞吐量场景

在此，我们会集中讨论索引吞吐量和速度的优化。一些是关于你每秒钟可以向 Elasticsearch 推送多少数据的，另一些是与索引无关的。

（1）批量索引

这个建议十分明显，但是你可能会感到奇怪，很多人都忘记了用批量索引代替逐个的索引文档。但是，需要注意，不要向 Elasticsearch 发送过多的超出其处理能力的批量索引请求。关注批量索引的线程池和它的大小（默认等于 CPU 核数，带有一个大小为 50 的队列），并尝试调整你的索引程序来匹配它们，否则，如果 Elasticsearch 不能处理它们的话，先是会是请求排队，然后你很快就会开始看到拒绝执行的异常，并且你的数据不会被索引。另一方面，记得不要让你的批量请求太大，否则 Elasticsearch 会需要大量的内存来处理它们。

为了举例说明，我会向你展示两种索引方式的区别。在第 1 张图中，展示的是正在逐个索引文档时的索引吞吐量。在第 2 张图中，我们索引同样的数据，但是不是逐个索引，而是每批 10 个。



如你所见，当逐个索引文档时，我们每秒能稳定地索引大约 30 个文档。在以 10 为批量使用批量索引时发生了变化，我们每秒大约能索引 200 个多一点的文档，所以可以轻易地看出它们的区别。

当然，这是索引速度的非常简单的对比，为了向你展示真实的区别，我们应该使用许多线程来让 Elasticsearch 达到它的极限。不过，关于使用批量索引对于索引吞吐量的提升，前面的对比应该给了你一个基本的概念。

(2) doc values 与索引速度的取舍

谈到索引速度时，我们不得不讨论 doc values。就如我们在本书中多次说过的，在 Elasticsearch 为了实现排序、聚合或分组，会需要反转字段，这需要巨大的内存，doc values 在这方面可以帮助我们。然而，记录 doc values 在索引时需要一些额外的工作。一方面，如果我们只关心最高的索引速度和最大的索引吞吐量，你应该考虑不使用 doc values。另一方面，如果你有大量的数据，为了使用聚合和排序功能而不产生内存相关的问题，使用 doc values 可能是唯一的办法了。

(3) 控制文档的字段

索引量的不同是有区别的，这很好理解。然而，这不是唯一的因素。文档的大小和对

其的解析同样重要。对于较大的文档，你不仅能够看到索引的增长，还能看到索引速度有所减慢。这就是为什么有时你会想要看看你正在索引和保存的字段。保持你存储的字段尽可能少，或者完全不使用它们，你在大多数情况下需要存储的字段是 `_source`。

除了 `_source` 字段，还有一件事，Elasticsearch 默认索引 `_all` 字段。提醒：`_all` 字段是 Elasticsearch 用来从其他文本字段收集数据的。在一些场景下，这个字段不被使用，这时最好关闭它。关闭 `_all` 字段十分简单，唯一需要做的是在类型映射中添加如下的内容：

```
"_all" : {"enabled" : false}
```

在索引生成阶段，可做如下处理：

```
curl -XPOST 'localhost:9200/disabling_all' -d '{
  "mappings" : {
    "test_type" : {
      "_all" : { "enabled" : false },
      "properties" : {
        "name" : { "type" : "string" },
        "tag" : { "type" : "string", "index" : "not_analyzed" }
      }
    }
  }
}'
```

减少文档的大小和其内文本字段的数量会让索引稍快一些。

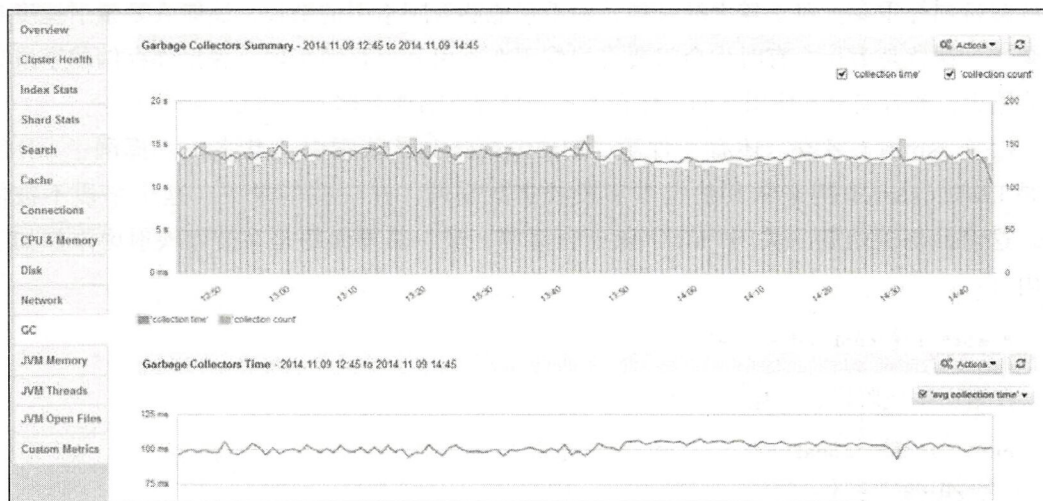
还有一件事，在禁用 `_all` 字段时，设置一个新的默认搜索字段是一个好的实践。我们可以通过设置 `index.query.default_field` 属性来指定。例如，对于我们的例子，我们可以在 `Elasticsearch.yml` 文件中设置它，并将设置为前面映射里的 `name` 字段：

```
index.query.default_field: name
```

（4）索引的结构和副本

在设计索引结构时，你需要考虑索引的分片和副本的数量，同时我们也需要考虑数据在 Elasticsearch 节点上的分布、性能、可用性、可靠性等。首先，把主分片部署到所有的节点上，于是我们可以并行的进行索引文档，这会加快索引的速度。

第 2 件事情是数据复制。我们需要记住的是过多的副本会导致索引速度下降。多个原因导致了这个问题。首先，你需要在主分片和副本间传递数据。其次，通常主分片和副本被部署在相同的节点上（当然不是主分片和它的副本，而是其他主分片的副本）。例如，我们来看看下图向我们展示了什么：



因为这个原因，Elasticsearch 需要主分片和副本的数据，而这会占用磁盘空间。取决于集群的配置，索引吞吐量在这类情况下会降低（取决于磁盘、同时索引的文档数量等）。

（5）调整预写日志

我们已经在 6.3 节中讨论过事务日志。Elasticsearch 有一个内部模块叫作 translog。它是一个基于分片的结构，用来实现预写日志（https://en.wikipedia.org/wiki/Write-ahead_logging）。基本上，它是用来让 Elasticsearch 向 GET 请求暴露最新的更新，确保数据的持久性，以及优化 Lucene 索引的写入。

Elasticsearch 默认在事务日志中保留最多 5000 个操作，或者最多占用 200MB 的空间。然而，如果我们想要获得更大的索引吞吐量，愿意付出数据在更长的时间内不能被搜索到的代价，我们可以调高这些默认值。通过配置 `index.translog.flush_threshold_ops` 和 `index.translog.flush_threshold_size` 属性（两者都是在索引级别生效并能通过 Elasticsearch 的 API 实时更新），我们可以设置存储操作的最大值数量和最大体积。我们曾经见到过把这个属性设置为默认值的 10 倍的情况。

需要记得，一旦发生故障，拥有大量事务日志的节点其分片的初始化会慢一些。这是因为 Elasticsearch 需要在 shard 生效前处理全部的事务日志。

（6）关于存储的思考

在应对高索引量的使用场景时，存储类型和其配置是一个关键点。如果你的公司能负担得起 SSD 硬盘，那么买吧。与传统的机械硬盘相比，它们具有速度上的优势，但是，当然是以更高的价格为代价的。如果你负担不起 SSD 硬盘，配置你的机械硬盘工作在 RAID0（<https://en.wikipedia.org/wiki/RAID>）模式，或者配置 Elasticsearch 使用多个数据路径。

另外，不要使用共享的或者远程的文件系统来保存 Elasticsearch 的索引，而是使用本

地的存储。远程和共享文件系统通常比本地磁盘慢，会使得 Elasticsearch 等待读写操作的完成，因而造成整体性能的下降。

（7）索引期间的内存缓存

还记得供索引缓存使用的可用内存越多（通过配置 `indices.memeory.index_buffer_size` 属性），Elasticsearch 在内存中容纳的文档就越多吗？当然了，我们不会让 Elasticsearch 占用 100% 的可用内存。默认使用 10%，但是，如果你真的需要一个较高的索引效率，你可以增加它。建议给每个在索引期间生效的分片分配 512MB 内存，但是，记住 `indices.memory.index_buffer_size` 属性是设置节点的，而不是分片的。所以，如果你给 Elasticsearch 20GB 的堆空间，且节点上有 10 个活动分片，那么 Elasticsearch 默认会给每个分片大约 200MB 内存用作索引缓存（20GB 的 10%，再除以 10 个分片）。

8.6 小结

在本章中，我们聚焦在 Elasticsearch 的性能和扩展，探讨了 doc values 是如何帮助我们提高查询性能的，垃圾回收器是如何工作的，以及在调整配置时需要关注什么。我们对查询做了基准测试，看到了热点线程 API 是什么样的。最后，我们讨论了在应对高查询量和索引量的使用场景时如何对 Elasticsearch 进行扩展。

在下一章中，我们会编写一些代码，创建一个 Maven 项目来实现一个 Elasticsearch 插件，编写一个自定义的 REST 行为来扩展 Elasticsearch 的功能。除此之外，还会学习为 Elasticsearch 引入新的分析插件需要做些什么，并且会创建一个这样的插件。

开发 Elasticsearch 插件

在上一章中，我们集中讨论了 Elasticsearch 集群的性能和扩展。看到了 doc values 如何帮助我们提升查询性能和降低查询的内存开销，这是以轻微降低索引速度为代价来实现的。我们查看了垃圾回收器如何工作的，以及在调整配置时需要关注什么。我们对查询做了基准测试，看到了热点线程 API 是什么样的。最后，我们讨论了如何对 Elasticsearch 进行扩展。到本章结束时，将涵盖以下内容：

- 如何配置开发 Elasticsearch 插件的 Maven 项目
- 如何开发一个自定义 REST 行为的插件
- 如何开发一个扩展 Elasticsearch 分析能力的自定义分析插件

9.1 创建Maven项目

在讲述如何开发 Elasticsearch 自定义插件之前，我们先探讨一下插件的打包方式。打包后的插件可以使用 plugin 命令安装到 Elasticsearch 中。为了实现这一目的，我们使用 Apache Maven (<http://maven.apache.org/>) 项目管理软件，它致力于使你的构建过程更简单、提供统一的构建系统、管理依赖等。



注意 本章是在 Elasticsearch 1.4.1 版本下编写并测试通过的。

另外请记住，你手中的这本书不是关于 Maven 的，而是讲述 Elasticsearch 的。我们会

尽可能少地讨论与 Maven 相关的信息。



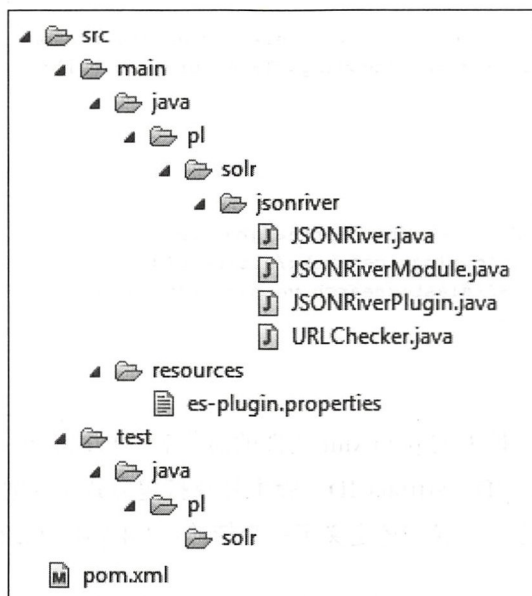
注意 Apache Maven 的安装是个简单任务。这里我们假定你已经安装好了 Maven。不过，如果你对安装步骤有疑问，请查询 <http://maven.apache.org/> 获取更多信息。

9.2 了解基本知识

Maven 构造过程的结果是一个 artifact。每个 artifact 由 ID、组织名、版本号来定义。这一点在使用 Maven 时至关重要，因为你将使用的每个依赖都由这 3 个属性唯一确定。

9.2.1 Maven Java 项目的结构

Maven 的理念非常简单。创建好的结构类似如下快照：



可见，代码被放到 `src` 文件夹下（普通代码在 `main` 文件夹下，而单元测试代码在 `test` 文件夹下）。尽管你可以调整默认布局，但 Maven 在默认布局下工作得更好。

9.2.2 POM 的理念

除了代码之外，你可以在图中看到一个位于项目根目录下名为 `pom.xml` 的文件。这是一个项目对象模型文件，可以描述项目本身、项目属性以及项目的依赖。不错，你不用手

工下载那些已经存在于某个可用 Maven 仓库中的依赖。在 Maven 工作过程中，它会自动下载并使用所需的依赖。你唯一需要在意的是，编写一个合适的 pom.xml 片段来告知 maven 需要哪些依赖。

比如，请看下面的 Maven pom.xml 文件：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>pl.solr</groupId>
    <artifactId>analyzer</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>analyzer</name>
    <url>http://solr.pl</url>

    <properties>
        <elasticsearch.version>1.4.1</elasticsearch.version>
        <project.build.sourceEncoding>UTF-8</project.build.
sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.elasticsearch</groupId>
            <artifactId>elasticsearch</artifactId>
            <version>${elasticsearch.version}</version>
        </dependency>
    </dependencies>
</project>
```

这是本章将要使用和扩展的 pom.xml 文件的简化版。可以看出，它以 project 根标签作为开始，然后定义了组织 ID、artifact ID、版本号及打包方式（本例中使用标准构建命令来打包成 jar 文件）。除此之外，我们还定义了一个依赖：1.4.1 版的 Elasticsearch 类库。

9.2.3 执行构建过程

为了运行构建过程，只需要简单地运行 pom.xml 所在路径中的命令：

```
mvn clean package
```

这个命令将启动 Maven。它将清除工作目录下所有自动生成的内容，编译代码，然后打包代码。当然，如果我们有单元测试，必须让它们运行通过才能打包。打包好的 jar 包将被写入 Maven 创建的 target 目录。



注意

如果想要了解更多 Maven 生命周期的信息，请访问 <http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>。

9.2.4 引入 Maven 装配插件

我们需要压缩插件代码来生成 zip 压缩文件。Maven 默认不支持纯粹的 zip 打包，为了支持这一功能，我们将使用 Maven 装配插件（可以在如下网址找到更多该插件的相关信息：<http://maven.apache.org/plugins/maven-assembly-plugin/>）。总的来说，这个插件可以把项目的输出和项目的依赖、文档、配置文件等聚合在一起，生成一个单独的归档文件。

为了让装配插件工作起来，我们需要在 pom.xml 中加入 build 片段。这个片段包含装配插件的信息、jar 插件（负责生成合适的 jar 文件）以及编译插件。指定编译插件的目的是让代码可被 Java 7 支持。除此之外，我们希望把档案文件放到项目的 target/release 目录下。pom.xml 文件的相关片段如下：

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <version>2.3</version>
      <configuration>
        <finalName>elasticsearch-${project.name}-${elasticsearch.version}</finalName>
      </configuration>
    </plugin>

    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>2.2.1</version>
      <configuration>
        <finalName>elasticsearch-${project.name}-${elasticsearch.version}</finalName>
        <appendAssemblyId>false</appendAssemblyId>
        <outputDirectory>${project.build.directory}/release</outputDirectory>
        <descriptors>
          <descriptor>assembly/release.xml</descriptor>
        </descriptors>
      </configuration>
      <executions>
        <execution>
          <id>generate-release-plugin</id>
          <phase>package</phase>
          <goals>
            <goal>single</goal>
```

```

    </goals>
  </execution>
</executions>
</plugin>

```

```

<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.7</source>
    <target>1.7</target>
  </configuration>
</plugin>
</plugins>
</build>

```

就近查看装配插件的配置，你会发现我们在 assembly 目录下设定了名为 release.xml 的装配描述符。这个文件将负责指示我们输出数据的文档类型。我们放到 assembly 目录下的 release.xml 文件的内容如下：

```

<?xml version="1.0"?>
<assembly>
  <id>bin</id>
  <formats>
    <format>zip</format>
  </formats>
  <includeBaseDirectory>false</includeBaseDirectory>
  <dependencySets>
    <dependencySet>
      <unpack>false</unpack>
      <outputDirectory></outputDirectory>
      <useProjectArtifact>false</useProjectArtifact>
      <useTransitiveFiltering>true</useTransitiveFiltering>
      <excludes>
        <exclude>org.elasticsearch:elasticsearch</exclude>
      </excludes>
    </dependencySet>
  </dependencySets>
  <fileSets>
    <fileSet>
      <directory>${project.build.directory}</directory>
      <outputDirectory></outputDirectory>
      <includes>
        <include>elasticsearch-${project.name}-
${elasticsearch.version}.jar</include>
      </includes>
    </fileSet>
  </fileSets>
</assembly>

```

同样，我们不需要了解所有细节，尽管知道正在发生些什么是一件好事。之前的代码将通知 Maven 装配插件把归档文件打包成 zip（<format>zip</format>）格式，并指出需要

排除 JUnit 和 Elasticsearch 这两个库（exclude 部分），因为它们已经在我们需要安装插件的 Elasticsearch 中提供了。此外，我们还指定需要包含我们项目的 jar 文件（include 部分）。

9.3 创建自定义REST行为

让我们从创建一个自定义 REST 行为来开始我们的扩展 Elasticsearch 之旅。我们选择它作为第一个扩展，是因为我们想要通过最简单的方式来介绍如何扩展 Elasticsearch。



注意

我们假定你已经如我们在 9.1 节中所做的那样创建了一个 Java 项目并使用 Maven。如果你想要使用现成可用的项目代码，请从 Packt 网站上获取本书的代码。

9.3.1 设定

为了演示如何开发一个自定义 REST 行为，我们需要了解它有怎样的功能。我们的 REST 接口真的非常简单，它会返回所有节点的名称或匹配传递给它的特定前缀的节点的名称。除此之外，它只在使用 HTTP 的 GET 请求时有效，所以，例如，POST 请求是不允许的。

9.3.2 实现细节

我们需要开发以下两个 Java 类：

- 一个类扩展 Elasticsearch 的 org.Elasticsearch.rest 包中的抽象类 BaseRestHandler，它负责处理 REST 请求，我们把它叫作 CustomRestAction。
- 一个类被 Elasticsearch 用来加载插件，这个类需要扩展 Elasticsearch 的 org.elasticsearch.plugin 包中的抽象类 AbstractPlugin，我们把它叫作 CustomRestActionPlugin。

除了以上两个，我们还需要一个简单的文本文件，在开发完我们提到的两个 Java 类后我们会讨论它。

1. 使用 REST 行为类

最有趣的类是那个我们用来处理用户请求的类，我们把它叫作 CustomRestAction。为了能够运转，它需要扩展 org.Elasticsearch.rest 包中的 BaseRestHandler 类，它是 Elasticsearch 中处理 REST 请求的基类。为了扩展这个类，我们需要实现 handleRequest 方法，在这个方法中我们会处理用户请求。还要实现一个有着 3 个参数的构造器，它会用来初始化基类和注册恰当的处理程序，这样我们的 REST 行为就可见了。

CustomRestAction 的完整代码如下：


```

public class CustomRestAction extends BaseRestHandler {
    @Inject
    public CustomRestAction(Settings settings, RestController
controller, Client client) {
        super(settings, controller, client);
        controller.registerHandler(Method.GET, "/_mastering/nodes", this);
    }
    @Override
    public void handleRequest(RestRequest request, RestChannel
channel, Client client) {
        final String prefix = request.param("prefix", "");
        client.admin().cluster().prepareNodesInfo().all().execute(new
RestBuilderListener<NodesInfoResponse>(channel) {
            @Override
            public RestResponse buildResponse(
NodesInfoResponse response, XContentBuilder builder)
throws Exception {
                List<String> nodes = new ArrayList<String>();
                for (NodeInfo nodeInfo : response.getNodes()) {
                    String nodeName = nodeInfo.getNode().getName();
                    if (prefix.isEmpty()) {
                        nodes.add(nodeName);
                    } else if (nodeName.startsWith(prefix)) {
                        nodes.add(nodeName);
                    }
                }
                builder.startObject()
                    .field("nodes", nodes)
                    .endObject();
                return new BytesRestResponse(RestStatus.OK, builder);
            }
        });
    }
}

```

(1) 构造器

对每一个自定义的 REST 类，Elasticsearch 在创建这类对象时都会传递 3 个参数：Settings 类型的对象，它包含配置信息；RestController 类型的对象，它用来绑定 REST 行为到 REST 端点；Client 类型的对象，它是一个 Elasticsearch 的客户，也是同 Elasticsearch 交互的入口点。这 3 个参数也同样是它的超类所需要的，所以我们调用基类的构造器来传递它们。

还有一件事：@Inject 注解。它允许我们告知 Elasticsearch 在创建对象期间把参数传入构造器。想了解这个注解的更多信息，请参见 Javadoc，你可以在 <https://github.com/Elasticsearch/Elasticsearch/blob/master/src/main/java/org/Elasticsearch/common/inject/Inject.java> 找到。

现在，让我们关注下面这行代码：

```
controller.registerHandler(Method.GET, "/_mastering/nodes", this);
```

这行代码的作用是注册我们自定义的 REST 行为，并且把它绑定到我们选定的端点上。第 1 个参数是 HTTP 方法类型，REST 行为会支持这个方法。就如我们前面说过的，我们只希望响应 GET 请求。如果我们想要响应多个 HTTP 方法，只需要对每个 HTTP 方法类型调用 `registerHandler` 方法。第 2 个参数指定我们自定义 REST 行为的确切端点。对于我们的例子来说，它是 `/_mastering/nodes`。第 3 个参数告诉 Elasticsearch 那个类需要负责响应我们定义的端点。对于我们的例子，它是我们正在开发的类，因此我们传递 `this`。

(2) 处理请求

尽管 `handleRequest` 方法是代码中最长的一个，但是它并不复杂。我们从读取参数的代码开始：

```
String prefix = request.param("prefix", "");
```

我们把 `prefix` 请求参数保存在名为 `prefix` 的变量中。默认的，在没有 `prefix` 参数传递给请求时，我们希望给 `prefix` 变量赋与一个空字符串值（默认值由 `request` 对象 `param` 方法的第 2 个参数定义）。

接下来，我们使用 Elasticsearch 的 `client` 对象获取 `NodesInfoResponse` 对象。`client` 对象可以执行 Elasticsearch 的管理命令。在这个例子里，我们可以使用异步的方式向 Elasticsearch 发送请求。不调用 `execute().actionGet()` 方法，这会一直等待直到有响应返回；而是调用 `execute()` 方法，它接收一个 `future` 对象作为参数，当查询结束时会通知这个 `future` 对象。于是，`handleRequest` 方法的其他代码就都在 `RestBuilderListener` 对象的 `buildResponse` 回调里。`NodesInfoResponse` 对象包含一个 `NodeInfo` 对象数据，我们用它来取得节点的名称。我们需要做的是返回包含指定前缀的所有节点，如果没有给出 `prefix` 参数则返回全部节点。为了实现这个效果，我们创建一个数组：

```
List<String> nodes = new ArrayList<String>();
```

我们使用接下来的 `for` 循环来迭代可用的节点：

```
for (NodeInfo nodeInfo : response.getNodes())
```

我们使用 `DiscoveryNode` 对象的 `getName` 方法来取得节点的名称，在 `NodeInfo` 对象上调用 `getNode` 方法可以得到 `DiscoveryNode` 对象：

```
String nodeName = nodeInfo.getNode().getName();
```

如果 `prefix` 参数是空的，或者节点名称以其开头，把这个节点名称添加到我们创建的数组中。当我们迭代完所有的 `NodeInfo` 对象，就开始构建响应，然后把它通过 HTTP 发送出去。

(3) 构建响应

关于 `CustomRestAction` 类的最后一件事情是处理响应，它是我们创建的 `buildResponse()` 方法的最后一部分代码的职责。这段代码非常简单，因为 Elasticsearch 已

经通过 builder 参数提供了恰当的响应构建器。它接收客户端调用时传递的 format 参数，所以我们以恰当的 JSON 格式发送响应，正如 Elasticsearch 默认的那样。我们也可以使用 YAML 格式 (<http://en.wikipedia.org/wiki/YAML>)。

现在，我们使用得到的 builder 对象来开始构造 response 对象（使用 startObject 方法），然后写入 nodes 信息（因为 nodes 的值是集合类型，它会被自动格式化为数组）。在 response 对象中创建了 nodes 属性，我们使用它来返回匹配的节点名称。最后我们使用 endObject 方法来结束 response 对象的构建。

在我们的对象准备好作为响应被发送后，我们返回 BytesRestResponse 对象。我们用如下代码来完成这个：

```
return new BytesRestResponse(RestStatus.OK, builder);
```

正如你所看到的，为了创建对象，我们需要传递两个参数：RestStatus 和 XContentBuilder。XContentBuilder 内保存着响应的内容，RestStatus 使得我们可以指定响应的代码，在我们的例子中是 RestStatus.OK，因为一切都很顺利。

2. plugin 类

CustomRestActionPlugin 类包含着 Elasticsearch 用来初始化插件的代码。它集成至 org.elasticsearch.plugin 包中的 AbstractPlugin 类。因为我们正在创建一个扩展，我们必须实现如下代码部分。

- ❑ constructor: 这是标准的 constructor，它接收一个参数，对于我们的例子，什么也不需要实现。
- ❑ onModule 方法：它包含了添加我们自定义的 REST 行为的代码，以便 Elasticsearch 能够知道这点。
- ❑ name 方法：插件的名称。
- ❑ description 方法：插件的说明。

整个类的完整代码如下：

```
public class CustomRestActionPlugin extends AbstractPlugin {
    @Inject
    public CustomRestActionPlugin(Settings settings) {
    }

    public void onModule(RestModule module) {
        module.addRestAction(CustomRestAction.class);
    }

    @Override
    public String name() {
        return "CustomRestActionPlugin";
    }
}
```



```
@Override
public String description() {
    return "Custom REST action";
}
```

constructor、name 和 description 方法都非常简单，我们略过不说，我们会聚焦在 onModule 方法上。这个方法接收一个参数：RestModule 类型的对象，它是使得我们可以注册自定义 REST 行为的类。Elasticsearch 会在所有可用的模块上调用 onModule 方法。我们做的仅仅是调用 RestModule 的 addRestAction 方法，传入我们的 CustomRestAction 作为参数。对于 Java 开发的部分来说到这里就结束了。

3. 向 Elasticsearch 声明自定义 REST 行为

我们的代码已经准备好了，但是我们还需要做一件事：我们需要让 Elasticsearch 知道注册我们插件的类是 CustomRestActionPlugin。为实现这个，我们在 src/main/resources 目录下新建一个 es-plugin.properties 文件，写入如下内容：

```
plugin=pl.solr.rest.CustomRestActionPlugin
```

我们只是指定了 plugin 属性，它需要赋值为我们用来注册插件的类（继承至 Elasticsearch 的 AbstractPlugin 的那个类）。这个文件会在构建过程中包含到 jar 文件里，Elasticsearch 在加载插件时会用到它。

4. 测试时间

当然了，我们可以到这里就结束了，但是我们不准备这样，我们想要向你展示如何构建每个插件、安装它，最后测试一下它是否能正常工作。让我们从构建插件开始。

（1）构建插件

我们从最容易的部分开始，构建插件。为了构建它，我们执行一个简单的命令：

```
mvn compile package
```

我们告诉 Maven 我们想要代码被编译和打包。在命令执行完毕后，我们会在 target/release 目录（假设你使用了与我们在本章开头描述的项目相同的配置）中发现包含插件的压缩包。

（2）安装插件

为了安装这个插件，我们会使用 Elasticsearch 发布包的 bin 目录下的 plugin 命令。假设我们的自定义插件包保存在 /home/install/es/plugins 目录中，我们会执行如下的命令（我们从 Elasticsearch 的根目录来执行）：

```
bin/plugin --install rest --url
file:/home/install/es/plugins/elasticsearch-rest-1.4.1.zip
```


我们需要在集群的所有节点上安装这个插件，因为我们希望在每个 Elasticsearch 上运行自定义的 REST 行为。



想了解更多关于安装 Elasticsearch 插件的信息，请参见我们前一本书《Elasticsearch Server, Second Edition》，或者查看 Elasticsearch 的官方文档，地址是 <http://www.Elasticsearch.org/guide/reference/modules/plugins/>。

安装完插件后，我们需要重新启动这些 Elasticsearch 实例。重启后，我们应该在日志中看到同下面类似的信息：

```
[2014-12-12 21:04:48,348][INFO ][plugins  
[Archer] loaded [CustomRestActionPlugin], sites []]
```

正如你所看到的，Elasticsearch 通知我们，一个叫作 CustomRestActionPlugin 的插件被加载了。

(3) 检验 REST 行为插件是否工作

终于，我们可以检验插件是否工作了。为了做到这个，我们会运行以下的命令：

```
curl -XGET 'localhost:9200/_mastering/nodes?pretty'
```

执行后，我们应该可以得到集群中全部的节点，因为我们并没有提供 prefix 参数。下面是 Elasticsearch 的响应：

```
{  
  "nodes" : [ "Archer" ]  
}
```

因为我们的集群只有一个节点，所以我们的节点数组中只有一个元素。

现在，我们试试如果添加 prefix=Are 到请求中会发生什么。我们使用的命令如下：

```
curl -XGET 'localhost:9200/_mastering/nodes?prefix=Are&pretty'
```

Elasticsearch 的响应如下：

```
{  
  "nodes" : [ ]  
}
```

如你所见，节点数据是空的，因为我们的集群中，没有任何节点的名称是以 Are 为前缀的。最后，我们测试另一种响应格式：

```
curl -XGET 'localhost:9200/_mastering/nodes?pretty&format=yaml'
```

现在响应不再是 JSON 格式了。看下一个由两个节点组成的集群其输出的内容：

```
---  
nodes:  
- "Atalon"  
- "Slapstick"
```

如你所见，我们的 REST 插件并不复杂，但是已经有一些功能了。

9.4 创建自定义分析插件

关于 Elasticsearch 自定义插件的最后一个话题是自定义分析插件。我们之所以选择展示自定义分析插件的开发过程，是因为这在某些情况下将非常有用，比如当你需要在公司项目中引入定制化的分析过程时，或者想要使用 Lucene 支持而 Elasticsearch 没有提供的分析器和过滤器时。由于创建一个分析器扩展相对之前的案例更加复杂，我们决定把它放到本章最后讨论。

9.4.1 实现细节

不管是从 Elasticsearch 自身视角来说，还是从需要开发的类数量来说，开发一个自定义分析插件是一件复杂的工作。相对上一个案例我们需要做更多的事。需要开发的任务如下：

- ❑ `TokenFilter` 类（来自 `org.apache.lucene.analysis` 包）的扩展实现。该实现命名为 `CustomFilter`，它将反转 `token` 的内容。
- ❑ `AbstractTokenFilterFactory`（来自 `org.Elasticsearch.index.analysis` 包）的扩展实现类。该实现类将向 Elasticsearch 提供 `CustomFilter` 实例。我们把它命名为 `CustomFilterFactory`。
- ❑ 自定义分析器，扩展自 `org.apache.lucene.analysis.Analyzer` 类，提供 Lucene 分析器的各项功能。我们称之为 `CustomAnalyzer`。
- ❑ `AnalyzerProvider`，扩展自 `AbstractIndexAnalyzerProvider` 类（来自 `org.Elasticsearch.index.analysis` 包），我们称之为 `CustomAnalyzerProvider`。它负责向 Elasticsearch 提供 `Analyzer` 实例。
- ❑ `AnalysisModule.AnalysisBinderProcessor`（来自 `org.Elasticsearch.index.analysis` 包）的扩展类。它将向 Elasticsearch 提供分析插件的名称。我们称之为 `CustomAnalysisBinderProcessor`。
- ❑ `AbstractComponent`（来自 `org.Elasticsearch.common.component` 包）的扩展类，它负责告知 Elasticsearch 使用哪个工厂类来创建自定义的分析器和过滤器。我们称之为 `CustomAnalyzerIndicesComponent`。
- ❑ `AbstractModule`（来自 `org.Elasticsearch.common.inject` 包）的扩展类，它将告知 Elasticsearch 为 `CustomAnalyzerIndicesComponent` 类生成一个单例。我们称其为 `CustomAnalyzerModule`。
- ❑ 最后是 `AbstractPlugin`（来自 `org.Elasticsearch.plugins` 包）的扩展类，我们称之为 `CustomAnalyzerPlugin`。

接下来让我们看看实现代码。

1. 实现 TokenFilter

本插件最有趣的部分是整个分析工作都是在 Lucene 层面完成的，我们所要做的仅仅是编写一个 `org.apache.lucene.analysis.TokenFilter` 扩展，我们称之为 `CustomFilter`。为了实现这个扩展，我们需要初始化基类并覆盖（`override`）`incrementToken` 方法。我们希望在 `CustomFilter` 类中实现反转 token 内容的逻辑。整个 `CustomFilter` 类的实现代码如下：

```
public class CustomFilter extends TokenFilter {
    private final CharTermAttribute termAttr =
        addAttribute(CharTermAttribute.class);

    protected CustomFilter(TokenStream input) {
        super(input);
    }

    @Override
    public boolean incrementToken() throws IOException {
        if (input.incrementToken()) {
            char[] originalTerm = termAttr.buffer();
            if (originalTerm.length > 0) {
                StringBuilder builder = new StringBuilder(new
String(originalTerm).trim()).reverse();
                termAttr.setEmpty();
                termAttr.append(builder.toString());
            }
            return true;
        } else {
            return false;
        }
    }
}
```

在这段代码中我们发现如下语句：

```
private final CharTermAttribute termAttr =
    addAttribute(CharTermAttribute.class);
```

该语句允许我们检索出目前正在处理的 token 的文本内容。如果要访问 token 的其他信息，则需要使用类似的其他属性（`attribute`），可以通过查看 Lucene 代码中 `org.apache.lucene.util.Attribute` 接口（http://lucene.apache.org/core/4_10_0/core/org/apache/lucene/util/Attribute.html）的实现类来弄清到底有哪些可用属性类。你现在需要了解的是，使用 `addAttribute` 静态方法可以绑定不同的属性类，以供我们在 token 处理阶段使用。

接下来是构造函数，它的实现很简单，仅仅用于初始化基类，因此我们略去不谈。

最后需要注意的是 `incrementToken()` 方法。如果 token 流中还有未处理的 token，该方法将返回 `true`，否则返回 `false`。我们首先要做的就是调用 `input` 对象的 `incrementToken()` 方

法检查 token 流中是否还有待处理的 token。input 对象是存储于基类中的一个 `TokenStream` 类型的实例。然后我们调用之前绑定好的属性的 `buffer()` 方法获取 term 文本。如果 term 拥有文本（文本字符串长度大于 0），则使用 `StringBuffer` 对象来反转文本内容，然后清除 term 缓冲区中的内容（通过调用属性的 `setEmpty()` 方法），再把反转后的文本追加到已经清空的 term 缓冲区中（使用属性的 `append()` 方法）。最后我们返回 `true`，因为反转后的 token 已经准备好接受进一步处理（在词项过滤器级别，我们不知道 token 是否还会被进一步处理，因此需要确保返回正确的值，以防万一）。

2. 实现 TokenFilter factory

`TokenFilterFactory` 是这个插件中最简单的类之一。我们需要做的仅仅是创建一个 `AbstractTokenFilterFactory` 类（来自 `org.Elasticsearch.index.analysis` 包）的扩展类，重写一个 `create()` 方法，并在该方法中创建我们自己的词项过滤器。代码如下：

```
public class CustomFilterFactory extends
AbstractTokenFilterFactory {
    @Inject
    public CustomFilterFactory(Index index, @IndexSettings Settings
indexSettings, @Assisted String name, @Assisted Settings settings)
{
    super(index, indexSettings, name, settings);
}
    @Override
    public TokenStream create(TokenStream tokenStream) {
        return new CustomFilter(tokenStream);
    }
}
```

可见，整个类非常简单。我们首先编写了构造函数，它将用于对基类进行初始化。然后，我们添加了 `create()` 方法，在该方法中使用由参数传入的 `TokenStream` 对象创建了我们自定义的 `CustomFilter` 类实例。

在继续之前，我们还想提及两件事情：`@IndexSetitngs` 和 `@Assisted` 注解。第一个会把索引配置信息以 `Setings` 对象的方式注入到构造函数中，当然这个过程是自动的。而被 `@Assisted` 注解标记的参数则会通过工厂方法的参数来注入。

3. 实现 analyzer indices component

我们希望本例的实现越简单越好，因此决定不在分析器的实现部分增加复杂性。为了实现一个分析器，我们需要扩展 Lucene 的 `Analyzer` 抽象类（来自 `org.apache.lucene.analysis` 包）。`CustomAnalyzer` 类的完整代码如下：

```
public class CustomAnalyzer extends Analyzer {
    public CustomAnalyzer() {
    }
}
```



```

@Override
protected TokenStreamComponents createComponents(String field,
Reader reader) {
    final Tokenizer src = new WhitespaceTokenizer(reader);
    return new TokenStreamComponents(src, new CustomFilter(src));
}
}

```



注意 如果你想要看看更复杂的分析器实现，请查看 Apache Lucene、Apache Solr 和 Elasticsearch 的源代码。

我们还需要实现 `createComponent()` 方法。该方法根据指定的字段（方法的第 1 个参数，String 类型）和数据（方法的第 2 个参数，Reader 类型）返回一个 `TokenStreamComponents` 对象（来自 `org.apache.lucene.analysis` 包）。在方法中，我们使用 Lucene 的 `WhitespaceTokenizer` 类创建了一个 `Tokenizer` 实例，该实例将按空格切分输入数据。随后我们创建了一个 `TokenStreamComponents` 对象，传入 `Tokenizer` 对象和 `CustomFilter` 对象，这样 `CustomAnalyzer` 就可以使用 `CustomFilter` 类了。

4. 实现 AnalyzerProvider

`AnalyzerProvider` 是本插件中除词项过滤器的工厂类之外的另一个 provider 实现。在这里我们需要扩展 `AbstractIndexAnalyzerProvider` 类（来自 `org.Elasticsearch.index.analysis` 包），以便于 Elasticsearch 创建我们自定义的分析器。实现代码非常简单，我们只需要实现一个用于返回分析器的 `get` 方法即可。`CustomAnalyzerProvider` 的代码如下：

```

public class CustomAnalyzerProvider extends
AbstractIndexAnalyzerProvider<CustomAnalyzer> {
    private final CustomAnalyzer analyzer;

    @Inject
    public CustomAnalyzerProvider(Index index, @IndexSettings
Settings indexSettings, Environment env, @Assisted String name,
@Assisted Settings settings) {
        super(index, indexSettings, name, settings);
        analyzer = new CustomAnalyzer();
    }

    @Override
    public CustomAnalyzer get() {
        return this.analyzer;
    }
}

```

我们首先实现了构造函数，用于初始化基类。然后创建了自定义分析器的单例，供 Elasticsearch 使用。请注意，该分析器类依赖于 Lucene 的 `Version` 类。因为我们自定义的这个分析器是线程安全的，一个单例可被反复重用，所以请不必担心。在 `get` 方法中，我们直

接返回已经创建好的分析器。

5. 实现 analysis binder

binder 是我们自定义插件的一部分，用于告知 Elasticsearch 分析器和词项过滤器的可用名称。CustomAnalysisBinderProcessor 扩展自 org.Elasticsearch.index.analysis 包中的 AnalysisModule.AnalysisBinderProcessor 类。我们覆写了两个方法。一个是 processAnalyzers，用于注册分析器，另一个是 processTokenFilters，用于注册词项过滤器。如果只有 Analyzer 或者只有词项过滤器，则只需要覆写其中的一个方法。CustomAnalysisBinderProcessor 类的代码如下：

```
public class CustomAnalysisBinderProcessor extends
    AnalysisModule.AnalysisBinderProcessor {
    @Override
    public void processAnalyzers(AnalyzersBindings
    analyzersBindings) {
        analyzersBindings.processAnalyzer("mastering_analyzer",
        CustomAnalyzerProvider.class);
    }

    @Override
    public void processTokenFilters(TokenFiltersBindings
    tokenFiltersBindings) {
        tokenFiltersBindings.processTokenFilter("mastering_filter",
        CustomFilterFactory.class);
    }
}
```

代码中第 1 个方法是 processAnalyzers()。它的输入是一个 AnalysisBinding 对象。该对象的 processAnalyzer 方法可以把我们自定义的分析器注册到指定名称下。注册时，需要传递一个可用名称和 AbstractIndexAnalyzerProvider 的实现类。该实现类负责创建分析器，在本例中，它是 CustomAnalyzerProvider。

代码中第 2 个方法是 procesTokenFilters()。它的输入是一个 TokenFiltersBindings 类对象。该对象的 processTokenFilter 方法可以把我们自定义的词项过滤器注册到指定名称下。注册时，需要传递一个可用名称和词项过滤器工厂的实现类，由该实现类来创建词项过滤器。在本例中，这个实现类是 CustomFilterFactory。

6. 实现 analyzer indices component

analyzer indices component 是一个节点级的组件。它允许重用分析器和词项过滤器。不过，在这里我们只想在索引级别重用我们的 analyzer，而不是在全局范围内（之所以要这样做只是为了演示）。我们需要扩展 org.Elasticsearch.common.component 包中的 AbstractComponent 类，子类命名为 CustomAnalyzerIndicesComponent。实际上，在子类中只需要实现一个构造函数。全部代码如下：

```

public class CustomAnalyzerIndicesComponent extends
AbstractComponent {
    @Inject
    public CustomAnalyzerIndicesComponent(Settings settings,
IndicesAnalysisService indicesAnalysisService) {
        super(settings);
        indicesAnalysisService.analyzerProviderFactories().put(
            "mastering_analyzer",
            new PreBuiltAnalyzerProviderFactory("mastering_analyzer",
AnalyzerScope.INDICES, new CustomAnalyzer()));

        indicesAnalysisService.tokenFilterFactories().put("mastering_filte
r",
            new PreBuiltTokenFilterFactoryFactory(new
TokenFilterFactory() {
                @Override
                public String name() {
                    return "mastering_filter";
                }

                @Override
                public TokenStream create(TokenStream tokenStream) {
                    return new CustomFilter(tokenStream);
                }
            }));
    }
}

```

首先，我们传递给父类必要的参数用于父类的初始化。然后我们使用如下代码片段创建了一个 CustomAnalyzer 类的分析器：

```

indicesAnalysisService.analyzerProviderFactories().put(
    "mastering_analyzer",
    new PreBuiltAnalyzerProviderFactory("mastering_analyzer",
AnalyzerScope.INDICES, new CustomAnalyzer()));

```

从代码中可见，我们使用 indicesAnalysisService 对象的 analyzerProviderFactories 获取了一个 PreBuiltAnalyzerProviderFactory 类（对象作为键值，对象名称作为键名）的 Map。我们向 Map 中放入一个新建的 PreBuiltAnalyzerProviderFactory 对象，命名为 mastering_analyzer。而为了创建这个 PreBuiltAnalyzerProviderFactory 对象，我们需要使用 CustomAnalyzer 和枚举值 AnalyzerScope.INDICES（来自 org.Elasticsearch.index.analysis 包）。AnalyzerScope 枚举变量的其他值还有 GLOBAL 和 INDEX。如果你想要在全局范围内共享分析器，可以使用 AnalyzerScope.GLOBAL。而如果想要为每个索引单独生成分析器，则需要使用 AnalyzerScope.INDEX。

我们使用相似的方法添加了自定义的词项过滤器。这里使用的是 indicesAnalysisService 对象的 tokenFilterFactories 方法，该方法返回一个 PreBuiltTokenFilterFactoryFactory 对象的 Map，对象名称作为 Map 的键名，对象本身作为键值。我们向 Map 中放入了一个名为

mastering_filter 的 TokenFilterFactory 对象。

7. 实现 analyzer module

analyzer module 非常简单。它扩展自 org.Elasticsearch.common.inject 包中的 AbstractModule 类。它的职责是告知 Elasticsearch，我们的 CustomAnalyzerIndicesComponent 类将作为单例来使用（对于这个类来说，使用单例就足够了）。它的代码如下：

```
public class CustomAnalyzerModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(CustomAnalyzerIndicesComponent.class).asEagerSingleton();
    }
}
```

从代码中可见，我们实现了一个简单的 configure 方法，该方法把 Custom-Analyzer-IndicesComponent 类绑定为一个单例。

8. 实现 analyzer plugin

analyzer plugin 是一个具体的 Elasticsearch 插件实现。它的职责是提供 Elasticsearch 关于插件自身的一些信息。它需要扩展 org.Elasticsearch.plugins 包中的 AbstractPlugin 类，并至少实现 name 和 description 方法。在这里，我们还实现了另外两个方法，用来注册我们的自定义分析插件。代码如下：

```
public class CustomAnalyzerPlugin extends AbstractPlugin {
    @Override
    public Collection<Class<? extends Module>> modules() {
        return ImmutableList.<Class<? extends
Module>>of(CustomAnalyzerModule.class);
    }

    public void onModule(AnalysisModule module) {
        module.addProcessor(new CustomAnalysisBinderProcessor());
    }

    @Override
    public String name() {
        return "AnalyzerPlugin";
    }

    @Override
    public String description() {
        return "Custom analyzer plugin";
    }
}
```

name 和 description 方法的职责显而易见，一个用来返回插件名称，另一个用来返回插件的描述信息。onModule 方法负责把 CustomAnalysisBinderProcessor 对象添加到由外部传

入的 `AnalysisModule` 对象中。

最后一个方法是 `modules`，我们目前还未接触过：

```
public Collection<Class<? extends Module>> modules() {  
    return ImmutableList.<Class<? extends  
Module>>of(CustomAnalyzerModule.class);  
}
```

我们在这里覆写了父类的对应方法，用于返回一个自定义插件所注册模块的集合。本例中，我们注册了一个单独的模块类 `CustomAnalyzerModule`，并返回一个拥有唯一入口的列表。

9. 把自定义分析器告知 Elasticsearch

一旦准备好所有代码，我们还需要做一件事：想办法让 Elasticsearch 知道，到底哪个类注册了我们的自定义插件 `CustomAnalyzerPlugin`。为此，我们在 `src/main/resources` 目录下创建一个 `es-plugin.properties` 文件。文件内容如下：

```
plugin=pl.solr.analyzer.CustomAnalyzerPlugin
```

我们在此只需要指定 `plugin` 参数。参数值为用于注册我们自定义插件的类（扩展自 `AbstractPlugin` 类）的名称。这个文件将在工程构建时被打包放进 `jar` 压缩包中，然后在插件加载过程中被 Elasticsearch 使用。

9.4.2 测试自定义分析插件

现在我们需要测试我们的插件，确保一切工作正常。为此，需要先构建好这个插件，然后把它安装到集群中的所有节点上，最后再使用 `Admin Indices Analyze API` 来验证它的运行情况。很好，着手去干吧！

1. 构建自定义分析插件

首先开始最简单的部分：构建插件。为此，执行如下命令：

```
mvn compile package
```

该命令让 Maven 编译并打包相关代码。命令执行完毕后，我们可以在 `target/release` 目录下找到打包好的文件（假定你使用的项目结构和我们在本章开始时描述的一样）。

2. 安装自定义插件

为了安装插件，我们需要如之前那样再次使用 `plugin` 命令。假定我们已经把插件包存放在 `/home/install/es/plugins` 目录下。执行如下命令来安装插件（我们在 Elasticsearch 主目录下运行本命令）：

```
bin/plugin --install analyzer --url  
file:/home/install/es/plugins/elasticsearch-analyzer-1.4.1.zip
```

我们需要在集群的所有节点上安装该插件，因为我们需要 Elasticsearch 能够在所有节点上找到我们自定义的分析器和过滤器，不管分析工作发生在哪个节点上。如果不这么做，肯定会遇到各种问题。



注意

如果想要了解更多关于 Elasticsearch 插件安装的知识，请参考前书《Elasticsearch Server, Second Edition》，或者阅读 Elasticsearch 的官方文档。

插件安装成功后，需要重启所有执行了安装操作的 Elasticsearch 节点。重启后，应该可以在日志中发现如下信息：

```
[2014-12-03 22:39:11,231][INFO ][plugins ]
[Tattletale] loaded [AnalyzerPlugin], sites []
```

这条信息表明，名为 AnalyzerPlugin 的插件已经成功地被 Elasticsearch 加载。

3. 检查自定义插件工作情况

最后我们需要检查一下自定义插件的工作情况是否和预期一样。首先我们创建一个名为 test 的空索引（实际上索引名称无所谓）。执行如下代码：

```
curl -XPOST 'localhost:9200/analyzetest/'
```

在此之后，我们使用 Admin Indices Analyze API（<http://www.Elasticsearch.org/guide/reference/api/admin-indices-analyze/>）查看我们的分析器是如何工作的。可执行如下命令：

```
curl -XGET 'localhost:9200/analyzetest/_analyze?analyzer=mastering_analyzer&pretty' -d 'mastering elasticsearch'
```

在响应中，我们应该可以看到两个被反转的 token：mastering,gniretsam 和 hcraescitsale（Elasticsearch 的反转表示）。Elasticsearch 的响应大致如下：

```
{
  "tokens" : [ {
    "token" : "gniretsam",
    "start_offset" : 0,
    "end_offset" : 9,
    "type" : "word",
    "position" : 1
  }, {
    "token" : "hcraescitsale",
    "start_offset" : 10,
    "end_offset" : 23,
    "type" : "word",
    "position" : 2
  } ]
}
```

可见，最终结果跟我们的期望一模一样。因此，看来自定义插件工作得与预期的一样好。

9.5 小结

在本章中，我们聚焦在开发 Elasticsearch 的自定义插件上。我们学习了如何建立合适的 Maven 项目来自动化构建 Elasticsearch 插件。我们了解到如何开发一个自定义的 REST 行为插件，最后，我们开发了一个包括自定义过滤器和分析器的插件，进一步扩展了 Elasticsearch 的分析能力。

我们已经到了本书末尾。有鉴于此，我们想要写一段简短的总结，向那些坚持阅读到最后的勇敢读者表达一下我们的心声。在出版《Elasticsearch Server, Second Edition》之后我们决定写作本书，我们觉得有太多的主题没有被覆盖，我们希望在本书中讨论它们。我们介绍了 Apache Lucene 和 Elasticsearch，并从 Lucene 索引和 Elasticsearch 两个角度探讨了查询和数据处理。我们希望此时你已经明白了 Lucene 的工作原理以及 Elasticsearch 如何使用 Lucene。希望你会觉得学习这个优秀的搜索引擎的旅程是值得的。我们讨论了当出现热点时会会有帮助的主题，例如 I/O 瓶颈、热点线程 API，以及如何缩短查询时间。我们还讨论了诸如根据应用场景来选择恰当的查询以及扩展 Elasticsearch。

最后，我们用两章探讨了 Java 开发：如何使用自定义插件来扩展 Elasticsearch 的功能。在本书的上一版中，我们还简单地描述了 Java API，但是我们认为这没有意义。API 值得专门写一本关于其使用的书籍，仅仅展示一些与之有关的东西是不够的。我们希望，你已经可以开发自己的插件，尽管我们并没有把所有可能的知识点都列出来。我们希望你自己找到所需的信息。

感谢你阅读本书。希望你能够喜欢它。希望它给你提供了一些你正在苦苦寻找的知识，并能够为你所用，不管你是在专业工作中使用 Elasticsearch 或者仅仅出于兴趣爱好而使用它。

最后，请时不时访问一下 <http://Elasticsearchserverbook.com/>。除了我们写的日常博客，我们还会发布一些没有加入本书或被删掉的内容片段，因为我们不想让本书变得无所不包。

作者简介

Rafał Kuć 资深软件开发专家，现任 Sematext 集团公司咨询专家及软件工程师。他专注于 Apache Lucene、Solr、Elasticsearch、Hadoop stack 等开源技术。他还是 solr.pl 网站的联合创始人，该网站致力于帮助人们解决 Solr、Lucene 的相关问题。

Marek Rogoziński 资深软件架构师和咨询师，专注基于开源搜索引擎（如 Solr、Elasticsearch 等）的解决方案及大数据分析技术（如 Hadoop、HBase、Twitter Storm 等）。他是 solr.pl 网站的联合创始人，除本书外，还著有《Elasticsearch Server》。

译者简介

张世武 毕业于北京科技大学数学系，曾先后在中科院计算所、新浪、汽车之家、国美大数据研究院等机构与公司从事搜索引擎研发和管理工作，在 Linux C/C++ 方面有着丰富的研发经验。感兴趣的研究领域包括全文索引内核、分布式搜索引擎框架、相关性排序、机器学习算法等。现任 BBD（成都数联铭品科技有限公司）北京研发中心技术经理，BBD 是一家行业领先的金融大数据公司，Elasticsearch 在 BBD 得到广泛的应用。

余洪淼 2002 年毕业于东北大学国际贸易专业，曾在汽车之家、去哪儿网从事研发、管理工作达 10 年之久，有丰富的网站架构、搜索引擎研发、项目管理经验，多年大型团队管理经验。

商旦 2007 年毕业于中国人民大学计算机专业，长期从事搜索引擎开发和调优工作，构建过多家知名互联网公司的站内搜索系统。先后任职于汽车之家、新东方，积累了丰富的网站架构和开发经验，熟悉 Linux 服务器运维。

Mastering Elasticsearch

Second Edition

Elasticsearch在美团有着广泛的应用，该搜索引擎框架功能强大，支持分布式，实时索引、搜索，插件丰富，使用简单便捷，是检索、广告计算、海量数据分析等领域的一把利器。本书第2版保持了一贯的品质，内容深入浅出，示例丰富，是大家进行Elasticsearch实践的必备资料。

—— 陈华良博士 美团广告技术负责人

很高兴看到本书第2版面市，Elasticsearch版本更新很快，新特性不断出现，新增内容很好地折射了Elasticsearch的变化。本书既有对底层技术的深入剖析，又有生动翔实的示例，能帮助读者快速提升在该领域的技术水平。

—— 高剑林 腾讯（架构平台部）资深技术专家

Elasticsearch是目前市场占有率最大的开源搜索引擎之一，国美线上搜索服务使用了该框架。该框架支持分布式，实时数据处理，容灾能力强，能灵活嵌入排序算法。本书第2版适时增加了很多新特性，同时又保留了第1版的精华内容，理论性与实践性结合得很好。

—— 宋洋博士 国美大数据研究院副总监

除了用于搜索，Elasticsearch也是日志存储、离线数据分析挖掘的利器。本书深入浅出，案例丰富，在信息检索模型、准实时搜索、分布式架构、系统优化等诸多方面都有精彩的论述。

—— 李伟博士 微软（bing）数据挖掘组高级工程师

尽管京东搜索引擎是自主研发的，但其架构原理与Elasticsearch有很多相似之处，如分布式、实时索引与检索、容灾处理、Ranking算法插件化等。本书对Elasticsearch的系统架构、底层技术原理有深入的阐述，非常适合中高级搜索引擎研发人员阅读。

—— 李洁 京东搜索与大数据部高级架构师

[PACKT]
PUBLISHING



投稿热线: (010) 88379604
客服热线: (010) 88379426 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

上架指导: 计算机/程序设计

ISBN 978-7-111-56825-4



9 787111 568254 >

定价: 79.00元